Aalto-yliopisto
Aalto-universitetet
Aalto University

Master's Programme in ICT Innovation

Anomaly detection for Linux system log

Rongjun Ma

Master's Thesis
2020

Aalto-yliopisto
Aalto-universitetet
Aalto University

Author Rongjun Ma

Title of thesis Anomaly detection for Linux system log

Programme Master's Programme in ICT Innovation

Major Human-Computer Interaction and Design

Thesis supervisor dr.ing. Gwenn Englebienne

Thesis advisor(s) M.Sc. (Tech.) Ossi Koivistoinen

Collaborative partner Nokia, Espoo

Date 01.11.2020 **Number of pages** 55 **Language** English

Abstract

The goal of this study is to develop effective methods for detecting anomalies in Linux Syslog collected during CI/CD deployment. The automatic detection will help improve developers' efficiency of debugging by saving much time that is spent on manually searching for errors in the sea of logs. For this purpose, two different types of anomaly detection methods are evaluated, namely workflow-based method and PCA-based method. During the experiment, different Natural language processing (NLP) methods such as word2vec and TF-IDF are tested for preprocessing and encoding the log message body. Long short-term memory (LSTM) and Principal component analysis (PCA) models are implemented separately as the representatives for the two types of methods mentioned above.

The experiment results of both methods turn out to surpass the performance of the baseline method stupid backoff, which is the current solution used by the thesis sponsor company. LSTM and PCA both reach a relatively balanced performance of recall and precision. As a harmonic indicator, the F1 score for PCA reaches 0.9043 and, for LSTM it is 0.9124, while the baseline is 0.6411.

In the conclusion section, different suitable use cases of different methods are discussed. These two methods proposed in this thesis contributes towards detecting syslog anomalies in an unsupervised manner when no label is provided.

Keywords anomaly detection, machine learning, CI/CD, LSTM, PCA

Preface

This paper is a report on my six-month master thesis research on Syslog anomaly detection. It originated from the idea by Nokia, ended up as a successful proof of concept to share with people who are interested in this topic. I am happy to see my proposals are valuable. Moreover, I have learned a lot.

I would very much like to thank my thesis supervisor Gwenn Englebienne who inspired me with effective methodologies, advisor Ossi Koivistoinen who guided me through every technical detail in this project, line manager Tommi Lundell who supported me with powerful machine and resources. I also want to thank my friends and family for all their support and help.

I wish this thesis work will also inspire you, who are reading my paper and enjoy reading.

August 12, 2020

Rongjun Ma

Abbreviation

ML machine learning

PCA principal component analysis

SVD singular value decomposition

LSTM long-short term memory

CI continuous integration

CD continuous delivery

NLP natural language processing

TF-IDF term frequency–inverse document frequency

GPU graphics processing unit

SPE squared prediction error

Contents

Chapter 1

Introduction

1.1 Background

Syslog records run-time information of system processes and it stores valuable data to help debug when a process fails or just to keep a record of issues. Anomaly detection, which aims at abnormal system behaviors by looking at Syslog data, allows developers to pinpoint and resolve issues in a timely manner. It plays a very important role in incident management, especially for large-scale distributed systems. Traditionally, developers inspect those logs manually with the use of keyword search (e.g., "fail", "exception") or regular expression match, which depends a lot on their domain knowledge and experience. However, such a manual process becomes inadequate when it comes to large-scale systems due to the following issues. There can be different practical challenges from system to system but three common challenges are the quantity of data, complex architecture, and tolerant mechanisms issues[12]. First, large-scale systems generate tons of logs. For example, in this study, the Linux system can generate 224,000 lines of log messages during the first two hours of testing one build. Second, under the modern development environment, a single developer is often responsible for sub-components and thus the whole system behavior can be too complex for one developer to interpret. Third, large-scale systems are built with different tolerant mechanisms and these may lead to a judgment of false positives. In practice, sometimes developers use regular expression to detect abnormal behaviors but it turns out to be log messages that are actually unrelated to the real failures[21]. To assist manual debugging, lots of work has been done such as developing the knowledge-sharing platform for developers to communicate and share similar issues. Furthermore, the development of Natural Language Processing and Machine Learning techniques also speed up the research on solving the problem via automatic anomaly detection.

Previous research on log anomaly detection works in three main directions, Principal Component Analysis(PCA) based methods, workflow-based methods, and invariant mining-based methods. As a brief introduction, PCA based methods try to create a normal space through learning the core features of normal encoded log entries and then calculate the distance from the to be tested logs to the normal space. Anomalies are identified by comparing the distance with a threshold. This method was first applied in this context by Wei et al. [33]. Second, workflow-based methods pay more attention to the flow of the process. The idea is to predict the possible

following logs based on previous ones and then comparing the actual log entry with prediction. The study by Du et al. [7] shares the same idea to detect anomaly by sequential predicting. The third one, invariant mining-based methods, proposes that logs are happening in pairs. For example, whenever there is an "open" in the log message, there will be a corresponding "close". Therefore, anomalies can be found based on pair-wise rules. Such rules are explained in the study by Lou et al. [23] Based on these studies, this thesis is aimed to develop a suitable model for detecting anomalies in Linux system logs provided by Nokia.

1.2 Thesis scope and objectives

The research topic for this thesis study is to find a suitable model to detect anomaly in the Linux system log. This section explains data source with a real-case scenario and also limits the scope of anomalies that will be targeted at in this study.

1.2.1 Data source and detection flow with real-case scenario

The real-case scenario of this anomaly detection tool will be to assist programmers in debugging during continuous integration (CI) and continuous delivery(CD) cycle. CI/CD is a practice that happens when new changes need to be merged to the main branch, the aim of it is to avoid conflicts and make sure the application is not broken by the commits. Graph 1.1 shows the pipeline of the CI/CD procedure at a high level. Whenever a developer merges some new changes, these changes are first validated by creating a build and running automated tests against that build. These tests include unit tests and integration tests. Then the commit will be continuously deployed to the quality assurance (QA) server to do the QA test. By running all the tests, it allows developers to merge changes continuously instead of waiting for release day and merge all the changes together, which often causes chaos. The intersection of CI and CD procedure is exactly where the anomaly detection tool helps, where commits are passed from CI and deployed to CD for the QA server test. Imagine when a developer submits a commit and it fails, which means some errors or conflicts happen with the commit, the QA server will return a system log recording execution process to the developer for debugging. Then the developer needs to go through the whole log file to find the issue, where anomaly detection results help by narrowing down the scope the developer needs to check.

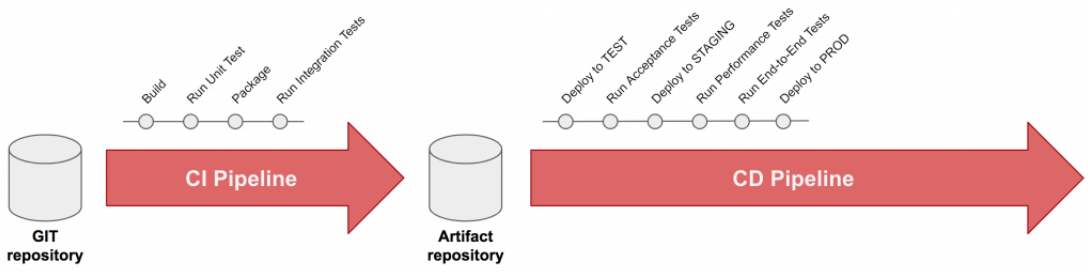

Figure 1.1: CI CD pipeline

During the QA server testing, all logs during deployment are continuously collected by the server and those which succeed going through testing will be the training sets for this study. The intuition behind is that even though new changes are committed continuously, the core structure remains the same for the same product. Thus, by learning the pattern of these successful logs, we can learn the workflow and distinguish those abnormal situations when the log violates the normal pattern. However, since it is still an updating procedure, how to choose data sets also influences the result of detecting anomalies for incoming new logs. For example, if all history logs are considered, the pattern learned might include noise from the antiquated model. Thus, how to balance between collecting enough data for training and keeping model up-to-date with the changing data is a factor to be considered in this experiment. In this study, the latest 19 log files are chosen to be the training set for the incoming new log considering the computational power and feedback from the end-user.

1.2.2 Types of anomalies to aim at

As logs are updated continuously during anomaly detection, the meaning of anomaly is not only to detect when it violates the normal pattern but also to help developers catch up with the latest changes. Therefore, anomalies aimed at in this study are divided into two categories.

One direction is to find anomalies that conflict with normal procedures. For example, suppose log A indicates the preparations for setting up an IP address and log B indicates operation with this IP address. Log A should always come before log B because only when the IP address is set up can the next operation being executed. If the order is reversed, the execution fails. In this situation, sequence $\{B, A\}$ violates the normal pattern so it is categorized as an anomaly.

The second direction is to detect a new pattern that is never seen with the learned model. For example, a new change is committed successfully which generates log C and it is never seen before. The new procedure will be $\{A, B, C\}$, and in this procedure, the updated C should also be detected as an anomaly.

1.2.3 Thesis structure

The remainder of the thesis proceeds as follows. In chapter 2 literature review is elaborated following the process of classical anomaly detection, from processing raw data to the detection stage. Chapter 3 describes the experiment designing, parameter settings, and also practical challenges. In chapter 4, experiment results are presented. Chapter 5 draws the conclusions and key insights from the experiment results. Lastly, chapter 6 discusses the limitations of this study and also future work.

Chapter 2

Literature Review

The procedure to detect anomalies usually consists of four phases: log parsing, log feature extracting, modeling, and anomaly detection. The first step, log parsing, is to transform raw log messages into a structured format so it can be modeled effectively by the machine. Then data mining techniques are applied to extract useful information from these logs for training, which is also part of data pre-processing work. In the third phase, the model will be developed and trained to learn patterns of normal workflows. Lastly, detect anomaly based on the learned knowledge.

2.1 Log parsing

A typical log message includes timestamp, hostname, and program name attributes that are followed by a free-form text, while only the text string is mandatory[22]. In this thesis, a normal log entry contains more information. As an example,


```
{"CURSOR" : "s=fd2a2a1c383d48d6b43a1bcda2be0248; i=5;
    b=7a568868e3854af698f75fca22e7a9e2; m=9c7508;
    t=59ecb36d2ebc3; x=71b6b5247db66e68",
"REALTIME_TIMESTAMP" : "1581970518895555",
"MONOTONIC_TIMESTAMP" : "10253576",
"BOOT_ID" : "7a568868e3854af698f75fca22e7a9e2",
"SOURCE_MONOTONIC_TIMESTAMP" : "0",
"TRANSPORT" : "kernel",
"SYSLOG_FACILITY" : "0",
"SYSLOG_IDENTIFIER" : "kernel",
"MACHINE_ID" : "4c926bd9584046d8a6564bbe44f74e3c",
"HOSTNAME" : "localhost",
"PRIORITY" : "6",
"MESSAGE" : "x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers' "}
```


is a complete log entry for this thesis in JSON format. As can be seen from categories (variables in capital letter), the log entry contains main body "MESSAGE"(free-from text) and other basic information[22], as well as some additional information customized by the program.

The purpose of log parsing is to extract the event template of each log message. For example, the logline mentioned above "*x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'* "will be parsed into the template with parameters represented by *

, like " *x86/fpu: Supporting XSAVE feature * : * * *".

To automate log parsing, many state-of-the-art algorithms have been proposed in recent years, such as iterative partitioning (IPLoM [24]), frequent pattern mining (SLCT [31], and its extension LogCluster [32]), longest common subsequence(LCS) (Spell[6]), parsing tree with fixed depth (Drain [11]). These methods share the same goal of parsing raw logs into templates but the intuitions behind them are quite different.

2.1.1 Iterative partitioning

As a representative of iterative partitioning method, IPLoM by Makanju et al. [24] works in the way that it partitions a set of log messages iteratively so that at each step the resulting partitions come closer to containing the same type of log entry. At the end of the process, it attempts to discover the line format in each partition and eventually the output of this algorithm is these discovered partitions and formats.

To be more detailed, IPLoM goes through four steps as follows:

1. Partition by token count. It is based on the assumption that log messages with the same format are also likely to have the same length so the algorithm first groups log messages with the same number of tokens together.

2. Partition by token position. This step is based on the assumption that the column with the least number of unique words is more likely to contain the constant words produced by the line format. For example, "*Connection from 255.255.255.255*", "*Connection from 0.0.0.0*" are in the same group after first step. In the first column, there is only one unique word "Connection", the same for "from". However, there are two unique values for the third position, "*255.255.255.255*" and "*0.0.0.0*". In this example, the positions of "Connection" and "from" have only one unique value, which means these two words are constant in this type of log entry. So, after the second step, the line format for these two entries will be "Connection from * ".

3. Partition by a search for bijection. This step aims to find a one to one relationship between two token positions, a summary of the heuristic would be to select the first two token positions with the most frequently occurring token count value greater than 1[24]. For example, 2.1 shows three types of log entries in one group after the second step. In this situation, "failed" has a 1-1 relationship with "on" and thus the connection of these two words is a bijection. However, there are also special cases of 1-M, M-1, and M-M relationships. For example, in figure 2.1 "has" shows a 1-M relationship with tokens "completed" and "been". Thus, a heuristic method is implemented to deal with those "M" relationships. With a ratio between the number of unique values in the set and the number of lines that have these values in the corresponding token position, a decision is made on whether to treat the "M" side as constant or variable values. In this example, "M" side refers to "completed" and "been", and a decision will be made whether "completed"/"been" is constant or not.

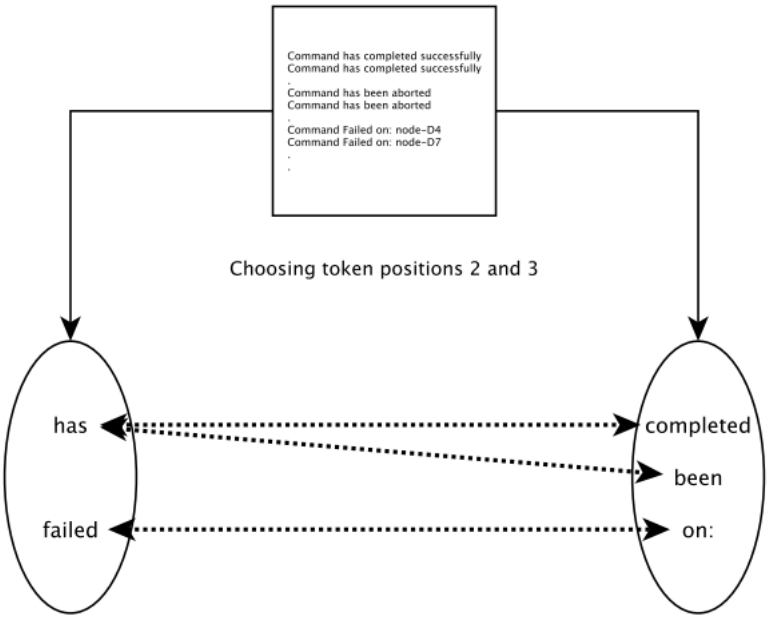

Figure 2.1: IPLoM step3: illustration about bijection[24]

4. Discover cluster descriptions (line formats) from each partition group. This step is just to summarize based on previous steps and eventually output the format of each partition.

2.1.2 Frequent pattern mining

The frequent pattern mining method aims to detect clusters that are observed in subspaces of the original data space by making a few passes over the whole dataset. It consists of three steps: First, make a pass over the whole dataset to get a summary of the data; second, make another pass to build cluster candidates; third, choose proper clusters from the candidates[31].

1. During the first step, the algorithm tries to summarize all frequent words. A word is considered to be frequent if it occurs (the position of the word in the line is also taken into account) at least N times in the dataset where N is a threshold defined by the user. After this step, dense 1-regions are created which is a collection of all the frequent words found.

2. This step generates cluster candidates by making another pass over the dataset. During the pass over, it finds all words that belong to dense 1-regions in each line. When one or more frequent words in dense 1 -regions are found in a line, a cluster candidate is formed. Then it will be added to a cluster candidate table through an if-else statement: add if it is new, ignore if it already exists. Take the log message "*Connection from 192.168.1.1*" as an example, suppose in this line, *(1, 'Connection')* and *(2, 'from')* are two words found. Then a region with the set of attributes *(1, 'Connection'),(2, 'from')* becomes a cluster candidate.

3. The last step is to inspect the candidate table and find all guaranteed cluster candidates to generate line formats. These guaranteed cluster candidates are selected based on a threshold value set by the user. Then line format will be generated for each of the selected candidates. For example, *(1, 'Connection'),(2, 'from')* corresponds to the line format "Connection from *".

2.1.3 Longest common subsequence

As can be learned from the name, Longest common subsequence(LCS) method works by finding the longest common subsequence of log entries. For example, log entry "*Connection from 192.168.1.1*" and "*Connection from 0.0.0.0*" share the words "connection" and "from", and the longest sequence between them is "*Connection from*". Graph 2.2 shows an basic workflow of Spell[6], which is a representative of LCS method. It parses log entries by the following steps. Before the journey of LCS, there are three variables to be identified .

LCSMap: a map that stores all status during the parsing procedure including new log entry and parsed line ID.
LCSSeq: a sequence that represents a line format of one type of log entry.
LCSObject: an object storing two parts of information, an LCSseq and all line Ids that match the LCSSeq.

1. There are two types of operations about LCSSeq, one is to update and the other one is to add new. Adding new LCSSeq is quite simple, if none of the existing LCSseq shares a common sequence that is at least half of the length of the given new log entry, then we create a new LCSObject for this new log entry and the sequence is the original log entry itself (e.g."*Temperature (41C) exceeds warning threshold*"). Then it comes to update when the next similar log entry comes (e.g. "*Temperature (43C) exceeds warning threshold*"). It will search all the LCSObjects until it finds a common sequence longer than half of it. In this example, it finds "*Temperature (41C) exceeds warning threshold*" but it disagrees with the *"(41C)*" part. Then it checks the length of both and updates "(41C)" with *. This example can be found in figure 2.2, the second box.

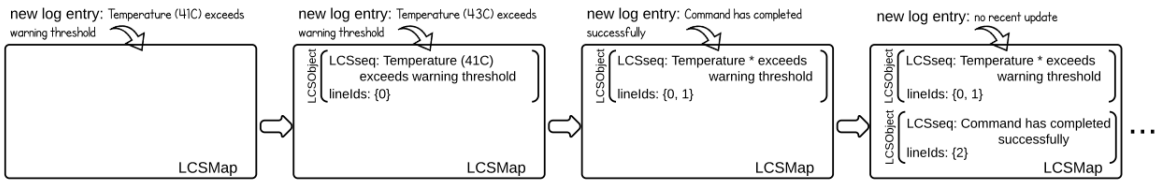

Figure 2.2: Basic workflow of Spell

2. The whole procedure of processing the raw logs works in this way. When a new log entry arrives, it is first parsed into a sequence of tokens. After that, it will be compared with all LCSObjects in the current LCSMap which stores all unique sequences. During the search, the rule determines if it matches or not based on length. There are two conditions to meet, first, it shares the common

sequence with the existing LCSSeq (including token position); second, length of the LCSsequence is greater than a threshold which is by default the length of original log entry sequence. If it meets both conditions, the line ID of it will be added to the corresponding LCSObject. Otherwise, it goes to the next step, create a new LCSObject.

For example, suppose we have an existing LCSseq: "*Connecting from **", where * represents variable and it is limited to a single token. Then, a new log entry "*Connecting from 0.0.0.0*" comes. First, this log entry has the same sequence as the existing LCSSeq. Second, the length of LCSSeq is 2 (tokens) and the length of this raw log entry "*Connecting from 0.0.0.0*" is 3, 2 is bigger than half of 3. Therefore, it meets both conditions and its line ID will be added.

2.1.4 Parsing tree with fixed depth

Drain [11], as a representative for paring tree with fixes depth, works by 5 steps. The same as Spell (the longest common sequence method), it works in an online mode. Graph2.3 shows a simple 2-layer tree model.

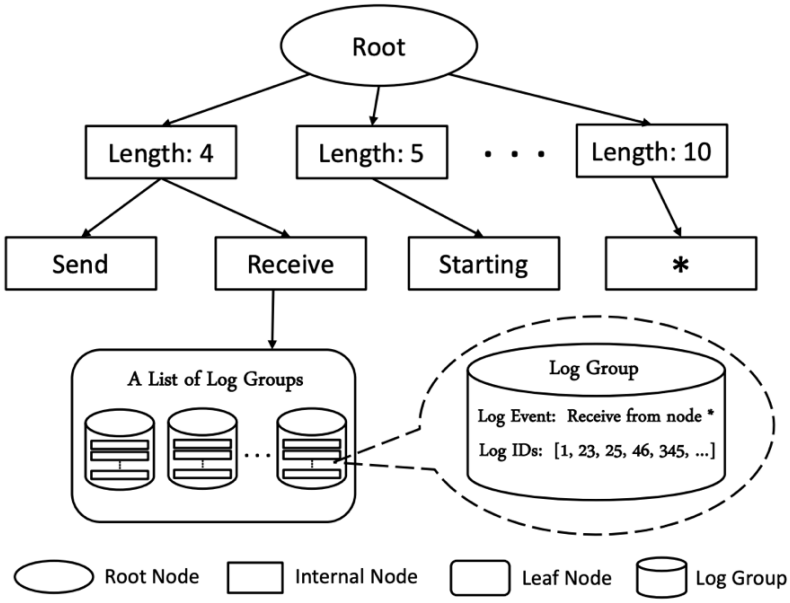

Figure 2.3: A tree model with depth of 3

1. According to an empirical study by He et al. [10], preprocessing helps improve parsing accuracy. Therefore, before employing the parsing model, Darin first removes some obvious parameters by the regular expression which can also be customized by users. Those obvious parameters are like block ID or IP addresses, which are just a string of numbers and characters generated automatically.

2. From this step, Drain starts to build the parsing tree with preprocessed logs from the first step. The 1-st layer nodes in the tree are based on the assumption

that log entries with the same length will be more likely to group together. Therefore, each node has a unique length(number of tokens in one log message) and parse logs by length.

3. This step supposes that the first word of a log message is usually constant. For example, the first word of *"Receive from node 4" "receive"* is considered a constant word. Then all unique first words will be one node in the 2-nd layer. However, in some cases, log messages can also start with a parameter. For example, *"120 bytes received"*. To avoid chaos due to these parameters, if the first position is a digit then it will be ignored and replaced by a *.

4. After step three, Drain gathered a list of log groups. Each group consists of log messages that start with the same word and have the same length. Suppose we now have a group of log messages starting with *"receive"* and have 4 tokens each message. In this step, they will be further distinguished by calculating the token similarity. The calculation is simply done by comparing each token in a specific position and set a threshold to judge if they are similar enough to be grouped together or not.

5. Last step is to take care of the updating issue. If logs find a suitable group in step 4, then the log ID of the current log message will be added to this group. Besides, the log event will be updated based on their difference (replace different parts by *).

2.1.5 Summary

To summarize the performance of different log parsing model, a comparison among these mentioned methods is illustrated in table 2.1. This table compares the performance of different log parsing techniques from five aspects. First, mode indicates the way that technique works. Typically, there are two types of modes – Offline and Online. Offline log parsing techniques work by batch processing and require all datasets to be available before parsing. However, online parsing techniques can handle log messages one by one in a streaming manner. The second one, coverage means the capability of a parser to parse all input logs. For Example, SLCT and LogCluster perform well by applying frequent pattern mining but fail when it comes to rare event templates. Third, preprocessing is a data cleaning step to remove some common variable values such as IP addresses and numbers, which requires manual work. Here, symbol "check" represents the preprocessing step is explicitly specified in a parser and "cross" means otherwise. Fourth, open-source indicates the current source code release status of those parsing methods. The last one, industrial use, indicates the practical value of these methods. Here, a "check" means the method is in industrial use and "cross" for pure research. The industrial value is evaluated according to the research by J.zhu et al.[35]

As summarized in the table above, different methods have different usage cases. They all process efficiently regarding time consumption but some of them can not handle rare log types. Also, they work in different modes, only Spell and Drain models have an online mode, which means they are able to parse logs in a streaming and timely manner. Considering the volume of logs is large in this study and it

Log Parser	Year	Technique	Mode	Coverage	Pre-processing	Open Source	Industrial Use
IPloM	2012	Iterative partitioning	Offline	✔	✘	✘	✘
SLCT	2003	Frequent pattern mining	Offline	✘	✘	✔	✘
LogCluster	2015	Frequent pattern mining	Offline	✘	✘	✔	✔
Spell	2016	Longest common subsequence	Online	✔	✘	✘	✘
Drain	2017	Parsing tree	Online	✔	✔	✔	✘

Table 2.1: Comparison between log parsing tools

also increases rapidly day by day, Spell and Drain models might be chosen to be applied in the parsing stage. Or, a better way of encoding the raw log message will be discussed in the experiment design stage.

2.2 Log extraction

After parsing the log, each log message now has a template with some positions of parameters (e.g. *"Receive from node * "*, * represents one parameter). However, there is still some other interesting information that has not been considered to be fed into the model, such as "TIMESTAMP", "PRIORITY". Some of this information is important because it might indicate the error, while some are always constant so it does not help with the anomaly detection. This log extraction step is to deal with the information and it is the last step of data preprocessing. The goal of this step is to construct feature vectors, which will be then fed into the model. In this step, appropriate variables are filtered to extract useful information, also related messages are grouped together because message groups show strong correlations among their members.[33] In a nutshell, all meaningful information will be considered to be as a part of the feature vectors.

The extracted information can be stored in separate matrices and trained separately. But it might also be different, which depends on the dataset and design of the model. As an example, in the model created by Xu et al.(2009)[33], the extracted information is stored in two matrices, namely the state ratio matrix and message count matrix.

In Xu et al.'s research, the state ratio matrix is used to capture the aggregated behavior of the system over a time window. Because in the dataset used by Xu et al., a large portion of messages contains state variables within countable categories. More importantly, these variables are closely related to anomalies. For example, in the dataset they used, the ratio between ABORTING and COMMITTING is very stable during daily execution but changes significantly when an error happens. Considering this situation, Xu et al. (2009) created a matrix to store the information of states by encoding the correlation: each row indicates the state ratio over a time window, while each column corresponds to a distinct state value. Graph 2.4 is an illustration of the ratio distribution and feature vector within a specific time window of 100 log lines. A complete state ratio matrix consists of lines of vectors like the one in graph 2.4, if one vector is judged as abnormal based on the algorithm, it indicates there must be something wrong within that time window. However, it can

not locate to a specific line, and in order to use this matrix, the log should be first divided into blocks based on the same time window (100 in figure 2.4).

COMMITTED	PREPARING	COMMITTING	ABORTED	Others
0.20	0.45	0.25	0.05	0.05

Figure 2.4: State ratio matrix

Another matrix introduced by Xu et al. (2009) is more widely used. It is composed of message count vectors, which describe the occurrences of basic log types. The log data for their experiment is collected from the Hadoop Distributed File System (HDFS). Files in HDFS are broken into block-sized chunks called data blocks. These blocks are stored as independent units and each block has a unique block ID [30], which is an important identifier for Xu et al[33]. Based on block ID, they divide the whole log into different blocks and encode information of each block to one row. As the final matrix, each vector represents one block. While dimensions of the vector correspond to all log template types across the whole log file, the value of each cell is the number of appearances of the message type in the corresponding block. Graph 2.5 shows the process of matching the raw logs with template id and then mapping it to the matrix. It is notable that in the third box of the message count matrix, only the first line corresponds to the example in the graph. The other two lines from block 326 and 327 are just dummy data to show how the matrix looks.

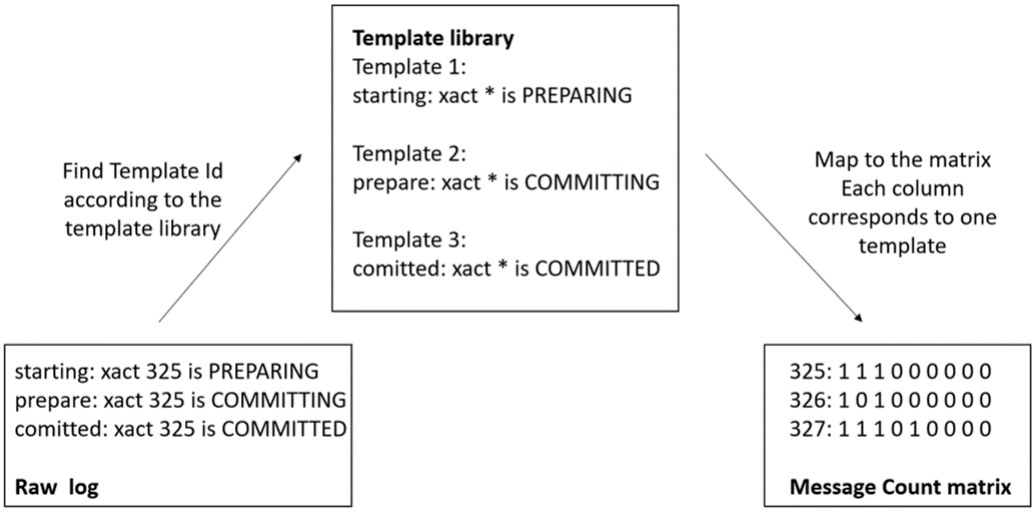

Figure 2.5: The process of constructing message count matrix

In Xu et al.'s study, they only consider the message body part because their dataset does not contain other info like "TIMESTAMP", "PRIORITY". So they construct

two matrices only based on the log parsing results (each raw log message returns with a template and a set of parameters (e.g. *"Receive from node * "*, [4])). Besides, they introduce two concepts, time window, and block. First, the time window is used to construct the state ratio matrix, which means grouping equal lines of log messages together and encoding the state ratio within the specific time window. By sliding the window, the complete state ratio matrix will be able to cover the whole dataset. For example, if the length of the window is set to be 100, then line 1 to line 100 will be encoded as row 1 of the matrix, line 101 to line 200 as row 2 and so on. Second, the block id groups log messages from one process together and it is used for constructing the message count matrix. By creating those meaningful subsets (each represents one process), it allows the algorithm to detect the specific process causing errors so it benefits debug a lot.

However, their matrix construction method seems to be not applicable to our situation. First, parameter values in our dataset hardly ever repeat and they are not so related to states. In fact, most of them are just machine ids or addresses which change from machine to machine. Therefore, it is not very meaningful in our case to follow the state ratio matrix construction. But our dataset does provide lots of additional information like "TIMESTAMP", "PRIORITY", a new way of encoding this information will be discussed in the following chapter – experiment stage. Second, in Xu et al. 's study, their dataset contains block id which helps distinguish different processes. But in our case, all processes log in parallel without indicators so it is hard to tell which process produces which lines of log. It might also affect the performance of the model because the order of log messages might be noise itself. Therefore, to construct a similar message count matrix, the time window technique might be a choice. In more detail, the matrix can be constructed based on the assumption that log messages within a specific time window come from the same process and will be encoded as one vector. The final encoding technique being used in the experiment will be discussed in the research design phase.

2.3 Modeling and detection

Currently, anomaly detection for log lines can be organized in three broad categories, PCA-based method[33] that distinguishes anomalies with a distance threshold to the normal space, workflow-based method[34] that captures illogical log lines, and invariant mining based method[23] which identifies co-occurrence patterns between different log messages.

2.3.1 PCA-based methods

Principal Component Analysis(PCA) is a statistical method that captures patterns of features by choosing a set of coordinates from high-dimensional data. Using the PCA technique, repeating patterns in features will be separated which makes it easier to detect abnormal situations. Figure 2.6 illustrates the intuition behind PCA-based anomaly detection method. Suppose there are now two variables in feature vector and they are plotted in this two-dimensional graph, S_d captures a strong correlation between these two variables and thus S_d is used to represent the normal situation of these two variables, which can be also called normal space. Then, two

new points A and B come. Intuitively, point A is far from the S_d which shows an unusual correlation so it is regarded as an anomaly. However for point B, even though it is far from most of the points, it still follows the pattern of S_d line and it is classified to the normal.

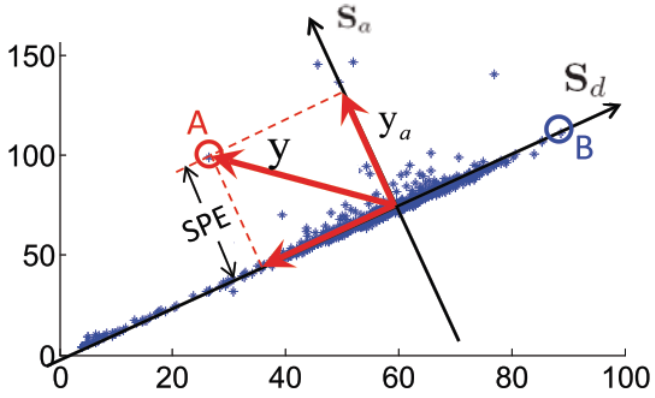

Figure 2.6: The intuition behind PCA detection with simplified data[33]

To explain the procedure with mathematical formulas, first a PCA model decomposes a vector into two portions[8],

$$x = \hat{x} + \tilde{x} \tag{2.1}$$

Here, in equation 2.1, $\hat{x}$ represents the modeled portion, which are components set chosen by PCA (normal space S_d in fig. 2.6), and $\tilde{x}$ corresponds to the residual portions (abnormal space S_a in fig. 2.6). The modeled portion is constructed through projecting original data with formula 2.2, where the $C = PP^T$ represents the projection matrix to transfer the original vectors into modeled space S_d of N dimensions.

$$\hat{x} = PP^T x = Cx \tag{2.2}$$

Suppose the original vector has M dimensions, then the left $\tilde{x}$ in equation 2.1 lies in the residual subspace of M-N dimension, where $\widetilde{C}$ represents the projection matrix on the residual subspace in equation 2.3.

$$\tilde{x} = (I - C)x = \widetilde{C}x \tag{2.3}$$

An unusual execution will be detected as an anomaly because it does not conserve the normal relations, which is shown in a way that increases its projection to the residual subspace S_a. As a result, the magnitude of $\tilde{x}$ reaches an extreme value that surpasses the threshold. Usually, a statistic for detecting these unusual conditions is the squared prediction error (SPE),

$$SPE = ||\tilde{x}||^2 = ||\widetilde{C}x||^2 \tag{2.4}$$

and the sample is considered normal only if,

$$SPE \leq \delta^2 \tag{2.5}$$

where δ^2 denotes a threshold for the SPE.

The choice of normal space S_d is based on how much information is chosen to be contained in S_d (e.g. choose k dimensions that explain 95% of variance to be S_d). The model will be different with different k values. But the intuition behind the PCA-based method is the same, to distinguish anomalies by examining the SPE of residual projection.

Moreover, there is another very important setting in this experiment about how to determine the threshold δ^2 . A statistical test for the residual vector known as *Q-statistic* developed by Jackson and Mudholkar[16] is used in many anomaly detection studies, such as in Xu et al.'s study of detecting large system problems [33] and Anukool et al.'s work of diagnosing network-wide traffic anomalies[19]. Q-statistic introduces the threshold under $1-\alpha$ confidence level as:

$$\delta_\alpha^2 = \phi_1[\frac{c_\alpha\sqrt{2\phi_2 h_0^2}}{\phi_1} + 1 + \frac{\phi_2 h_0(h_0-1)}{\phi_1^2}]^{\frac{1}{h_0}} \tag{2.6}$$

where

$$h_0 = 1 - \frac{2\phi_1\phi_3}{2\phi_2^2}, and\ \phi_i = \sum_{j=r+1}^{m} \lambda_j^i; for\ i = 1,2,3 \tag{2.7}$$

In equation 2.6 and 2.7, λ_j is the variance captured when the data is projected on the $j-th$ principal component, c_α is the $1-\alpha$ percentile in a standard normal distribution, m and r describe the shape of the projection matrix where m stands for the rows of data and r denotes the number of normal axes. Additionally, as pointed out by Jensen and Solomon [17], the Q-statistic changes little even when the distribution of the original data differs substantially from Gaussian distribution. Thus, Q-statistic can be widely used for PCA-based anomaly detection regardless of the data distribution.

With two feature matrices explained in section 2.2, Xu et al. (2009) build two PCA models for two types of anomalies, event occurrence anomaly and parameter anomaly respectively (See figure 2.7). The output of their model is an array composed of binary values 0 or 1, with 1 representing anomalies. Each value in the output array corresponds to one vector of the input matrix. Thus the length of output array matches the number of input vectors, which is exactly the number of blocks. It means it can only tell if one block is abnormal, but it can not distinguish which specific line is abnormal. In the last step, they developed a decision tree visualization to summarize the PCA detection results in an intuitive picture that is more friendly to operators because the judgment rule and threshold are then visualized.

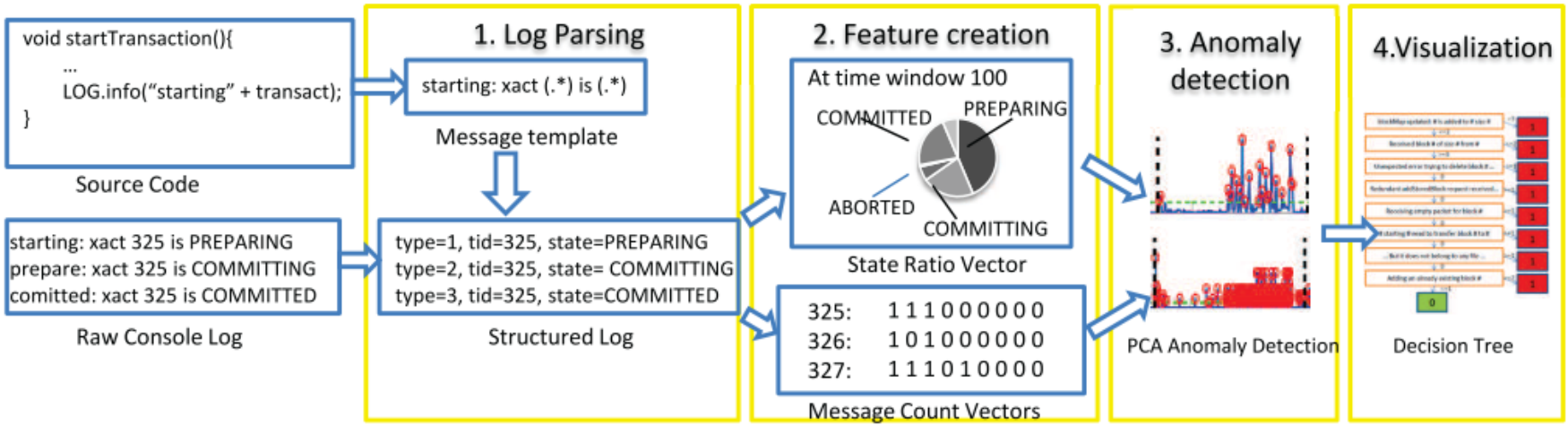

Figure 2.7: The workflow of PCA-based method by Xu et al.[33]

2.3.2 Workflow-based methods

System logs are usually produced following a set of rigorous rules and there is always a workflow pattern for a certain program[7]. This feature determines how workflow-based methods work: First learning from regular workflow and meanwhile constructing workflow models with the learned knowledge. Then detecting outliers that deviate from the sequence model. There can be various ways to learn the pattern. For example, in Deeplog [7], Du et al. proposes a method to model sequences of log entries based on Long Short-Term Memory(LSTM) model. Additionally, they demonstrate a way to make decisions in a streaming fashion. Figure 2.8 shows an overview of the Deeplog architecture.

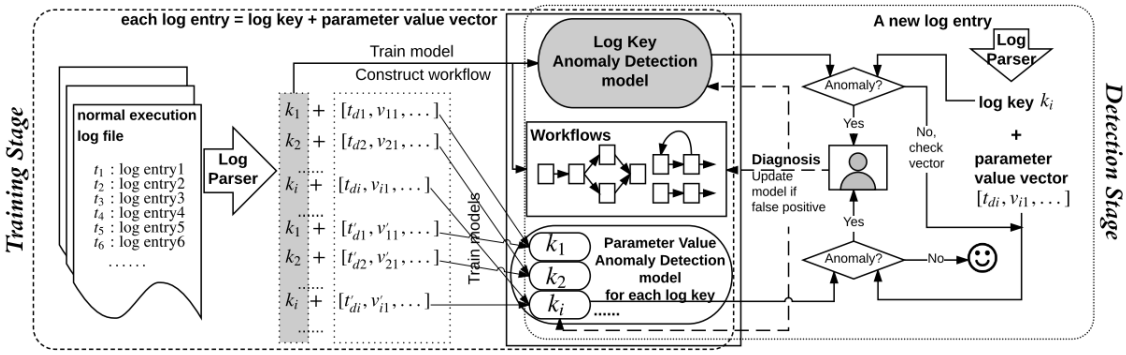

Figure 2.8: Deeplog architecture[7]

In this model, log keys and parameters are extracted and stored in two separate matrices. Also, they are processed by two separate models. In this method, either the log key or the value vector of its parameters is predicted as abnormal will lead to the result that the log entry being marked as an anomaly. In Deeplog's design, LSTM is applied to deal with the log key anomaly detection. Figure 2.9 illustrates a detailed view of the stacked LSTM model being used. Given a sequence of log messages, LSTM model is trained by learning the probability distribution $P_r(m_t = k_i|m_{t-h}, ..., m_{t-2}, m_{t-1})$ ($k \in K$, K represents all log keys) of the next log key. By finding logs keys that maximize the probability P_r, a set of possible log keys are predicted. Then, by comparing the predicted ones with the real log key that actually happened, the anomalies will be distinguished (if the real log key is out of the predictions).

As illustrated in Figure 2.9 : m_{t-h} represents the input. Together with cell state vector (C_{t-i}), the output from last block (H_{t-h}) work as hidden neurons. They both influence the output of the current block and then will be passed to the next block to initialize its state. All these operations are accomplished through a set of gating functions, which determines state dynamics to control how much previous information to be retained.

However, system behavior may change over time and the training data may not always cover all possible normal execution patterns. Therefore, Deeplog [7] also creates a mechanism to update model weights with manual feedback. For instance, suppose the model is predicting based on the previous 3 log entries $\{k_1, k_2, k_3\}$ and

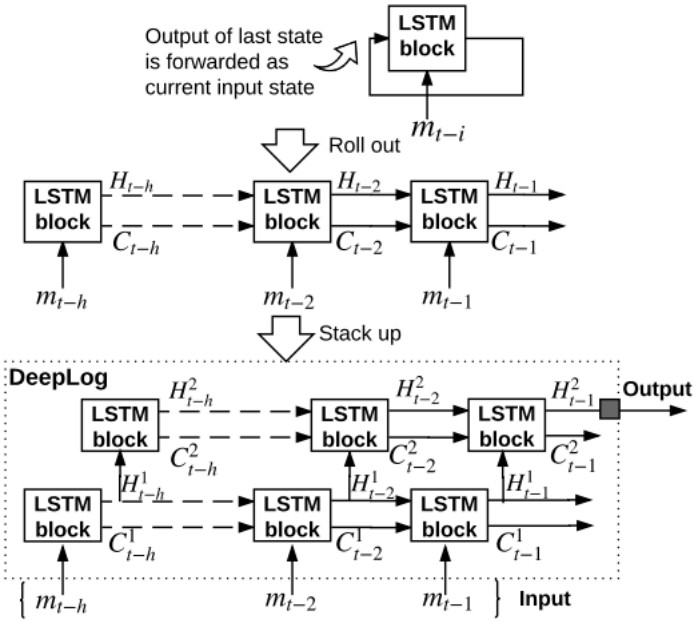

Figure 2.9: A detailed view of statcked LSTM model[7]

it predicts the next one to be k_1 with a probability of 1, while the next one appears to be k_2, so k_2 is labeled as an anomaly. However, if a user reports that this prediction is a false positive which means k_2 is actually normal but it is classified as an anomaly, then the model will update the weight of its model by learning the new pattern $\{k_1, k_2, k_3 \rightarrow k_2\}$. In this way, the model is able to learn continuously and benefit even with execution updating.

2.3.3 Invariant mining based methods

Invariant mining methods are based on linear relationships between logically pairwise log messages (e.g. *"open file"*, *"close file"*) from console logs that can be learned with automatic techniques. It was first applied to log anomaly detection in the study of Lou et al.[23]. Linear relationships are extracted from system execution behavior, and thus, they always carry the logic rule of the workflow. As a simple example for invariant: in normal executions of a system, if a file is opened then at some stage it should be closed. In a way that can be calculated, the log messages indicate *"Open file"* should be equal to the number of logs that indicate *"Close file"*. Suppose message indicating "open file" is type A, and *"close file"* is type B, a rule is then created in a mathematical way:

$$c(A) = c(B) \tag{2.8}$$

When invariant rules like this are developed, an anomaly will be identified when a log message breaks certain invariants. In this sense, it not only detects the anomalies but also makes the errors logically explainable.

The workflow of this method is: Firstly, parsing log messages into structured logs.

Then, grouping them based on set of rules to identify the groups of cogenetic parameters. Thirdly, counting messages and mining invariant using greedy algorithm to obtain invariant candidates and validate them using collected historical logs. Finally, detecting anomalies based on these invariant rules. See graph 2.10 for illustration of invariant mining workflow.

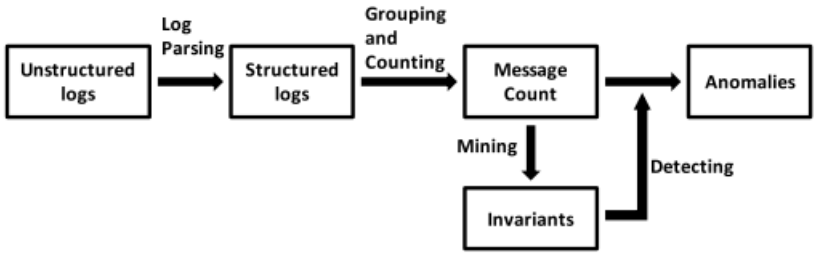

Figure 2.10: The workflow of invariant mining technique for log anomaly detection

This method is suitable for dataset that shows strong correlations between log pairs. However, in our dataset it is hard to capture this kind of pair-wise relationships. The dataset used in this study is collected during installing a software on distributed systems, the amount of the same type of messages differ from system to system and it does not follow certain invariants sometimes. For example, during the initializing stage there are a lot of messages about initializing certain devices that are detected (e.g. *"Initializing XFRM netlink socket"*). For different machines, devices that needs to be initialized might be different (e.g. some machines does not have certain devices) and thus the amount of this type of messages differ a lot. Moreover, these types of message usually happen without *"closure"*. It is hard to capture invariant rules in our case and therefore, this method is not further researched in this study.

2.4 Baseline

The baseline for experiments in this study is the model that is currently used by the company. The current model is called *Stupid Backoff* method that combines multiple *N-gram* models in a very simple way[25]. First, the next log template label to be predicted is denoted as i , the maximum n-gram length as k and the probability distribution of some j-gram model as $P_j(t_i|t_{i-j+1}, ..., t_{i-1})$. In the inference phase, the procedure works as follows:

1. Given the sequence of prior log templates $t_{i-k+1}, ...t_{i-1}$, if it has occurred in the training data, return $p_k(t_i|t_{i-k+1}, ..., t_{i-1})$.

2. If not, check if the sequence $t_{i-k+2}, ...t_{i-1}$ (corresponding to the (k-1)-Gram model) was in the training data and return $p_{k-1}(t_i|t_{i-k+2}, ..., t_{i-1})$ if so.

3. If not, continue in the same way until the lowest-order model is reached. If none of the models can provide a prediction, return a probability of zero.

There are, however, drawbacks with the current solution: Firstly, it does not give an actual probability distribution, since each of the submodels has their own distributions. This makes it harder to interpret results and improve the analysis. Secondly,

no smoothing is done so that any predictions based on low counts are probably bad. Thirdly, especially in the specific context of anomaly detection for our dataset, the fact that some log keys/templates never come after a given sequence can actually be significant, which means they have zero probability to appear. However, since the model goes back off, it can then wrongly back off to a lower n-gram submodel and give a nonzero probability to it.

Chapter 3

Research design

3.1 Data analysis

3.1.1 An overview of Log data

The log data for this study is quite different from log data experimented in previous researches mentioned in literature review. Table 3.1 shows some rough statistics about log files. Additionally, some key insights from the log datasets are described as follows:

Variables	value
Number of log data files	20
Rows per file	240,000
Log types in total for 20 files (Spell)	7,730
Train data	80%
Validation data	10%
Test data	10%

Table 3.1: Comparison between log parsing tools

key insight 1: Sequence is important
These log files show strong patterns in the message sequence. By plotting the log distribution of each log file, it is observed that these log files all show a similar pattern. This sequence pattern will be a good indicator to detect problems rather than just considering single messages as independent instances. Figure 3.1 is a scatter plot of the log type (Event ID) distribution, which only plots the data from five log files for the simplicity. In this graph, the X-axis refers to the index of log lines and the Y-axis refers to event ID which Indicates the log template that specific line belongs to, different colors represent different log files. As can be seen from the graph, many of the dots are overlapping which means these logs share the same pattern. In the meanwhile, in the upper part of the plot, it shows some differences between different logs. These differences with bigger event ID correspond to new types of log entries. These are the log entries that have never be seen by the log template library because with adding log templates, the event ID number grows. These new entries are also anomaly targets of this study.

Figure 3.1: Log type distribution

Some previous work(e.g.[17][30]) divides log messages into blocks, then construct vectors of log frequency and detect when log counts appear to be abnormal. This method can trace the sourcing process causing errors. But it might lose the logical information of sequence. For example, suppose in a normal block log type 1 appears twice and log type 3 appears four times in the order of {1,3,1,3,3,3}. By using the log count matrix, it will be able to detect when log type 1 appears 4 times and type 3 only once as an anomaly because the quantity changes. However, it will not tell when the order changes. For example, {3,3,3,3,1,1} might also be an anomaly but since the quantity keeps the same, it will not be recognized by the log count matrix. Second, in our dataset, it is hard to differentiate log entries from different processes.

key insight 2: Noisy data
Log rows in the dataset can originate from multiple parallel threads running at the same time, which means the order of some log entries can differ from log to log. This makes detection based on the sequence more difficult. However, as discussed in key Insight 1, these log files still show a similar pattern of workflow from a global perspective. So, how to capture the sequence pattern and at the same time avoid noise can be a challenge in this study.

key insight 3: Uneven distribution
Log types in the dataset are very unevenly distributed. For example, some log templates such as logs at the beginning of the files don't repeat almost at all. Meanwhile, some log types repeat quite often and make up a large portion of the whole file. For example, the most frequent log type (log template: *node controller - * type * msg audit (*.* *) **) makes up 23% for the whole log files in the experiment dataset. Figure 3.2 plots the distribution of different log types, the x-axis is the log id (After log parsing, each log line finds its template and thus has an assigned template id), while the y-axis represents the times of occurrence of that specific type of log. As can be seen from the graph, logs with template id 588 reach a peak, which appears

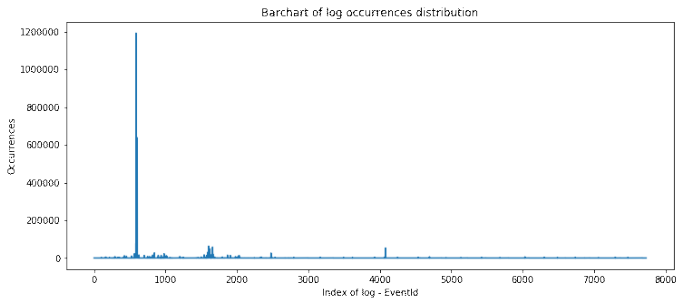

Figure 3.2: Distribution of different log templates

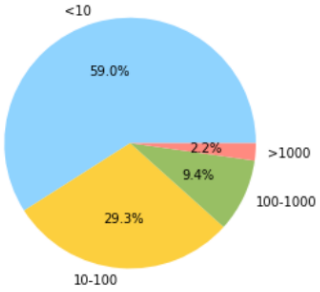

Figure 3.3: Ratio of occurrences

1,192,072 times out of 4,535,115 log lines in total. By contrast, most of the log types appear less. 3.3 plots the ratio of occurrences of log messages. 59.0% of log templates only appear less than 10 times through the whole training data set and 29.3% appear 10-100 times. These two groups make up 80% percent of the log templates.

3.1.2 Data selection

Raw log files for this study contain lots of information and not each log line contains the same attributes. By going through the whole dataset, it is observed that each log line at least contains 12 basic common attributes, while the most can have 23 attributes. Based on the analysis, these 12 categories are namely *"CURSOR"*, *"REALTIME TIMESTAMP"*, *"MONOTONIC TIMESTAMP"*, *"BOOT ID"*, *"SOURCE MONOTONIC TIMESTAMP"*, *"TRANSPORT"*, *"SYSLOG FACILITY"*, *"SYSLOG IDENTIFIER"*, *"MACHINE ID"*, *"HOSTNAME"*, *"PRIORITY"* and *"MESSAGE"*. According to discussions with experienced engineers about and data analysis on the dataset about data variance and importance, some of these columns are dropped. Among all these 12 columns of basic information, four categories are considered to be valuable for anomaly detection and thus chosen as data to be the training set for our model. These four categories are interpreted as follows:

Monotonic Timestamp: The monotonic timestamp indicates the time when one log entry is received by the journal. It is a relative timestamp and formatted as a decimal string in microseconds. It begins from zero as the beginning of each log file and then increases as time goes by. Compared to real-time timestamp, which is formatted as the clock time and thus different for every log file, it better describes at which time point is this log most likely to come.

Transport: Transport indicates how the entry is received by the journal service, valid transport is: *Audit* – for those read from the kernel audit subsystem; *Syslog* – for those received via the local Syslog socket with the Syslog protocol; *Kernel* – for those read from the kernel, etc. Transport is helpful to detect these kinds of anomalies that certain types of message come from the wrong source, or the source identifier appears at a wrong time.

Priority: Priority indicates the severity level of log entries. The levels are from 0-7, while 0 is the most severe level that means emergency which might be a "panic"

condition, level 7 is debugging info that is only useful for debugging.

Log message: Log message is the main body of information that contains the real content of the log. How to encode this part is a very important task that affects the model performance. In the literature review section, all the parsing and extracting techniques are applied exactly for the log message body.

3.2 PCA experiment design

3.2.1 Data pre-processing

PCA is a technique that reduces the dimension by keeping enough principle components that contain most of the information. In this study, there are four variables to be expressed in feature vectors, namely *"Monotonic Timestamp", "Transport", "Priority", "Log message"*. To include all the information in feature vector, instead of using log entry count in separate blocks to construct vectors, a new way of encoding these four categories for each line of logs is proposed here. In other words, each log line will be encoded as one vector in the input matrix, which makes it possible to associate anomaly with a specific line. To construct the feature vector, a detailed discussion about encoding each variable is as follows:

Monotonic Timestamp: Since monotonic timestamp is a relative time, it can be used to describe the sequence. However, instead of using absolute accurate microseconds, the percentage might be more appropriate to construct vectors. Using the percentage of time range instead of the exact time point compensates for differences in execution speed on different forms of hardware and data noise. As an example of data noise, suppose log A and log B are from two processes working in parallel, sometimes A's timestamp is later than B's while sometimes it is not. But anyhow they will both happening around a percentage of the log because the processes are still executed in a high-level order. In this case, the noise can be solved by rounding the timestamp percentage. To get the percentage, we can easily calculate the duration from zero points divided by the total duration of executing and omit a few decimal places. Also, this can be tuned during experiments. As an instance, suppose the duration of one log file is 155,155,485 microseconds, and log A comes at 608,000 microseconds. Divide 608,000 by 155,155,485 we get 0.003918649..., keep five decimal places we will then have 0.00392 which is 0.392% and it means that this log type A comes roughly around 0.392% of the log regarding the time of logging.

Transport: Transport is the variable that indicates how the log entry is received by the journal service, it describes the journal source by hardware at a high level. There are six types of valid transports, namely *"audit", "driver", "Syslog", "journal", "stdout"* and *"kernel"* which represent different sources for logging. The *"audit"* tag marks logs read from the kernel audit subsystem; The *"driver"* tag is for internally generated messages; *"Syslog"* for those received via the local Syslog socket with the Syslog protocol; The *"journal"* tag is for those received via the native journal protocol; The *"stdout"* is for those read from a service's standard output or error output; The *"kernel"* tag is for those read from the kernel. Transport can not tell which specific process produces log entries but it indicates the source of the log at a higher

level. Sometimes errors happen when the wrong transport produces some abnormal messages, so it is interesting to include transport in the training dataset. Since this variable is categorical data, it will be encoded via one-hot encoding, which is a technique to map categorical data into binary vectors. For example, after processing the transport part of row n that comes from *"audit"* will be:

Row number	audit	driver	syslog	journal	stdout	kernel
n	1	0	0	0	0	0

Table 3.2: An example of transport encoding

Priority: Priority between 0 to 7 is compatible with the Syslog priority concept and it indicates the emergence level of a specific logline. Unlike the way to encode transport which is nominal data, priority is ordinal data so the value of number matters. Therefore, the value is kept for priority. But considering 0 represents emergency information which is more severe than 7 debug information, the value for priority is reversed. To be more specific, 0-7 is mapped to 7-0.

Log message: The log message is the main body for the whole training, which is the most important part. The goal of this experiment is to detect anomalies that are narrowed down to a specific line number, unlike the PCA experiment done by Xu et al. [33] which detects the abnormal blocks. Therefore, each line of the log will be encoded as one vector to construct the whole matrix. There are two ways to encode it, one is to be the same as done in other researches using template id only, the other way is to use word embedding techniques to encode by tokens. About the first method to use log id, it is proved and widely used by most of the research of log anomaly detection. But unlike other studies that construct the log count matrix divided by blocks, we are encoding each log line separately. In this situation, a single number representing a log message is too simple and it does not carry interpretable meaning for the PCA model. This is also why the second way using word embeddings is proposed. Word embedding is a technique that enables words to be mapped to numerical vectors with vocabulary learned from the whole set of text information. This creates semantically more meaningful dimensions for PCA to model. For example, if log messages are coming from the same process, they are supposed to have some similar words. By catching these keywords, PCA is able to recognize when a wrong log message happens at the wrong time. As a conclusion, a word embedding is chosen to encode the message body.

Word Embedding is a collective name for a set of language modeling and feature learning techniques. In more detail, there are two typical types of mainstream word embedding techniques. The first one is based on word frequency and another type is prediction-based embedding. Two representatives are "Term frequency-inverse document frequency" (TF-IDF) and Word2vec, respectively. The Word2vec model is a two-layer neural network that is trained to reconstruct linguistic contexts of words. It relies on either skip-grams or continuous bag of words (CBOW) to create neural word embeddings. This method is good to capture relationships between words and linguistic context. But in our case, it is not very useful to look into linguistics and the most important task is to capture the keywords. Therefore, TF - IDF method

is chosen to be the encoding method for log messages.

By the definition of TF-IDF, it is a metric that multiplies the two quantities TF and IDF. Term frequency (TF) is a direct estimation of the occurrence probability of a term showing up in the document at hand. It represents how often a term occurs in the documents , and therefore how representative it is of the document. Inverse document frequency (IDF) can be interpreted as "the amount of information" according to conventional information theory [3][18]. It models how common the term is in other documents, and therefore how unique and informative it is in general. To explain the calculating process, suppose a corpus consists of only two documents, as shown in graph 3.4. Take word "sunny" as an example, TF is simply the occurrence

Doc 1
Today is a sunny day.

Term	Term count
Today	1
is	1
a	1
sunny	1
day	1

Doc 2
Friday is a rainy day.

Term	Term count
Friday	1
is	1
a	1
rainy	1
day	1

Figure 3.4: An example of tf-idf word embedding

probability of the term so:

$$TF(\text{“}sunny\text{”}, d1) = \frac{1}{5} \tag{3.1}$$

$$TF(\text{“}sunny\text{”}, d2) = \frac{0}{5} = 0 \tag{3.2}$$

IDF is constant per corpus, it calculates the ratio of documents that include the word "sunny" through dividing the number of documents at hand in total (N) by the number of documents the word "sunny" appears (d). In this example, "sunny" only appears in document 1 out of 2 documents in total so:

$$IDF(\text{“}sunny\text{”}, D) = log(\frac{N}{d}) = log(\frac{2}{1}) \approx 0.301 \tag{3.3}$$

Therefore, by multiplying TF and IDF, the TF-IDF value of word "sunny" is 0.0602 for document 1 and 0 for documents 2. Following the same equation, it can be calculated that the TF-IDF scores of "is", "a", "day" are all 0. As can be learned from the process, TF-IDF is trying to assign a higher value to those informative words while excluding these frequent words that appear everywhere. It is because of this feature, TF-IDF can be an effective way to catch keywords in our dataset as it is expected. In this study, TfidfVectorizer from the Scikit-learn library is used for the encoding work. Due to the diversity of words, if the whole vocabulary of around 8000 words is kept, the matrix will be too large. And the disk and memory space will not be able to allocate enough space to store the matrix. So in this study, only the top 3000 words with the highest TF-IDF score are kept after vectorizing.

To summarize the data pre-processing section, after data processing a complete feature vector (One row of the matrix) looks like example 3.3. In this table, the first row indicates column name and the numbers of columns corresponding to that specific variable are marked inside the bracket.

Mono timestamp(1)	Transport(6)	Priority(1)	Log message(3000)
0.9925332	0,0,0,0,0,1	6	0.33298,0,0,..,0.22324,0,0,0,0

Table 3.3: An instance of complete feature vector

3.2.2 Modeling

Modeling has four steps, the first two-step can be understood as the training stage while the latter two steps are implementing the predicting function. The first step is to decompose the original input matrix. In this study, singular value decomposition (SVD) is used for feature ranking and selecting, which is the goal of PCA. The intuition behind is that any matrix can be decomposed into the production of three separate matrices as illustrated in equation below, Where A is a random matrix, U is an orthogonal $m \times m$ matrix, V is an orthogonal $n \times n$ and S is a real diagonal $m \times n$ matrix. The elements of the leading diagonal of matrix S are called singular values, by ranking these elements we will be able to get principle components in order.

$$A = USV^t \tag{3.4}$$

Second, select the first k principal components that contain 95% (95% is the default value which is normally used in PCA dimension reduction, it is also used in the research by Xu et al.) of the information in the original dataset. Third, extract the transform matrix from the second step and project test dataset with the transform matrix to the same high-dimensional space. The last step is to detect anomalies by calculating the projection to the residual subspace discussed in section 2.3.1, where the threshold is calculated based on the c_α value selected (corresponding to confidence level). The equation for calculation is illustrated in section 2.3.1 equation 2.6 and the lookup table for c_alpha is listed in the appendix . The whole procedure is illustrated in graph 3.5.

3.2.3 Model tuning and evaluation

In this PCA experiment, two important parameters can be modified. The first one is the percentage of information to keep, it determines the dimension of the projection matrix. As a default, PCA usually takes 95%. During the experiment, it can be adjusted until it performs best in distinguish anomalies. The second parameter that can be changed is *c_alpha*, it determines the confidence level of the detecting result because it is used to calculate the threshold. According to the lookup table for *c_alpha*, a suitable confidence level can be chosen so it is not too strict that it detects everything as anomalies or too loose that it detects nothing.

About the evaluation process, because our dataset does not provide labels that

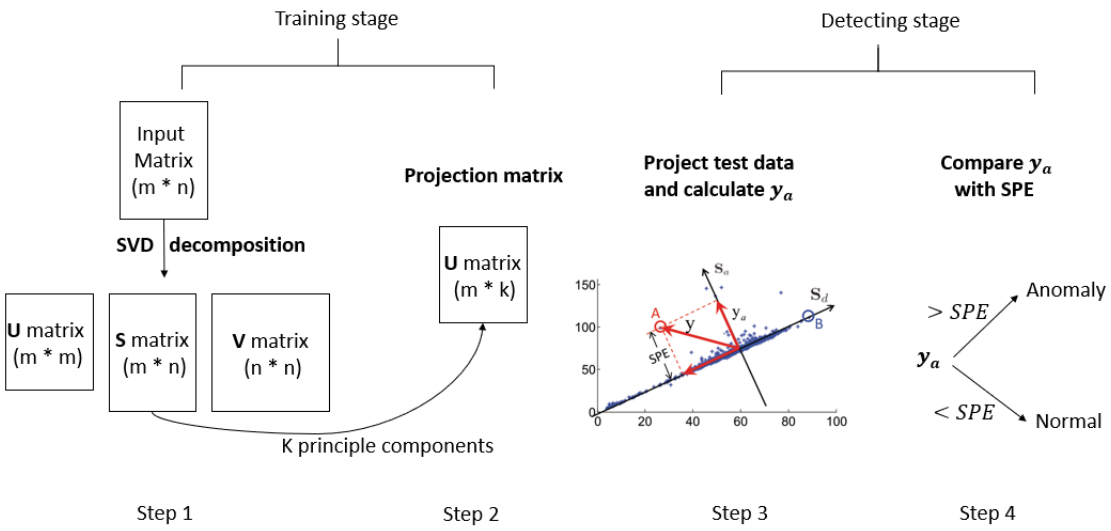

Figure 3.5: PCA-based anomaly detection modeling workflow

indicate anomalies or not, it is hard to calculate the confusion matrix. Therefore, a way to fake errors as anomalies manually and label these lines as anomalies are used here. The test data set for this study is a successful log with 240,000 lines. In this test log, 100 lines will be modified manually to inject errors. This error injection is based on empirical study, including domain knowledge provided by experienced engineers about frequent anomalies. Then the values of True Positive(TP), True Negative(TN), False Positive(FP) and False Negative(FN) are counted to calculate standard metrics of the detection. In this study, precision, recall, and F-measure are used as measurements.

Recall, or sensitivity denotes the ratio of real positive cases that are detected correctly[28]. In this study, recall measures how sensitive is the model to anomalies. It is defined by the equation below:

$$Recall = \frac{TP}{TP + FN} \tag{3.5}$$

Precision, or confidence, is the proportion of predicted positive cases that are true positives. Precision describes the rate of discovering real positives, anomalies in this study. It is defined by the equation:

$$Precision = \frac{TF}{TP + FP} \tag{3.6}$$

F-measure is the harmonic mean of the two metrics above, namely recall and precision. F1 score is a single measure to capture the effectiveness of a system. It can be calculated with the equation:

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.7}$$

In statistical hypothesis testing, the rejection of a true null hypothesis (False Positive) is defined as type I error. Conversely, type II error is the non-rejection of a false

null hypothesis (False Negative)[5]. In different fields of research, the seriousness of type I and type II error varies. For example, if the goal is to detect cancer, then the type II error is relatively more acceptable. Because it is likely that the momentary stress of a false positive is better than failing to treat the disease at an early stage. But in most fields of study, type I error is seen as more serious than type II errors. The reason is that with type I error, the null hypothesis is wrongly rejected, and eventually, it leads to a conclusion that is not true. In this study, type II error is also more acceptable because the anomaly detection goal is to assist manual debugging by narrowing down the anomaly scope and if False Positive happens, it can still be excluded by humans in the later stage. However, if potential anomalies are not detected, some important details might be ignored.

3.3 LSTM experiment design

3.3.1 Data pre-processing

LSTM anomaly detection is based on sequential predicting, it predicts the next log based on previous ones. Then by comparing the predictions with the real following one, anomalies are identified with the rule that if a real log is among predictions then it is normal and vice versa. In order to learn the sequence pattern, a supervised learning model is created which uses each following log id as the label for the previous sequence. To construct the feature matrix, if we regard log types as categorical data, then the prediction is to predict the following possible category. For example, in $\{k1, k2, k4- > k2\}$ it uses a sequence of k1, k2, k4 to predict the next possible one, where the result is k2. To represent logs by category, the result from log parsing (template id) can be directly used. Because the main focus of the LSTM model is on the sequence of messages, other variables like timestamp, priority, transport used in the PCA method are not considered here. Take a window size of 10 as an example, one row of the matrix will be a sequence of 10 log ids, and the corresponding label for the row will be the next log id. Eventually, a feature matrix of template ids is constructed as shown in table 6.3. In this table, numbers like "67" do not carry any

Feature matrix(10)	Labels(1)
66,67,67,67,67,67,67,67,67,68	69
67,67,67,67,67,67,67,67,68,69	69
67,67,68,69,69,69,69,69,69,69	70
67,68,69,69,69,69,69,69,69,70	71
...	...

Table 3.4: An instance of feature matrix for LSTM model

meaning with the value itself but it is just meaning a category. To avoid bias for the model, one-hot encoding is used here also to encode these categories as nominal binary vectors. After transforming, the matrix looks like table 3.5. Because there are 7730 log types in total, there are 7730 * 10 columns for the input matrix and 7730 columns for the label matrix.

Feature matrix(77300)	Labels(7730)
0,0,0,0,0,1,0,0,0...	0,0,0,0,0,1,0,0,0...
0,0,0,0...1,0,0,0...	0,0,0,0...1,0,0,0...
0,0,0,0...1,0,0,0...	0,0,0,0...1,0,0,0...
...	...

Table 3.5: An instance of one-hot encoded feature matrix for LSTM model

3.3.2 Modeling

In this study, the initial LSTM model is built with Keras using TensorFlow back end[2]. The main model consists of only two hidden layers, 128 neurons for each layer, and between each layer there is a dropout layer to avoid overfitting. The initial model architecture and some other parameters are illustrated in table 3.6. Besides, "Categorical_crossentropy" is used as a loss metric in the model. "Categorical crossentropy" is a loss function that is used in multi-class classification tasks. These tasks are usually when the model needs to decide which category an instance belongs to. Formally, this loss function is designed to quantify the difference between two probability distributions. In the LSTM model, the output will be a probability distribution over 7730 log template classes for each sequence. By minimizing the loss value between the predicted distribution and real vector, this model will learn to predict the right template for the following log by assigning a high probability to the right template. Graph 3.6 shows how the output of the model looks like. The categorical crossentropy loss function calculates the loss of an instance by the equation below:

Layer	Neurons	Input shape	Output shape	Parameter
Input layer	128	(77300,)	(128,)	initializer='orthogonal'
Dropout layer				dropout percentile = 0.2
Hidden layer	128	(128,)	(128,)	
Dropout layer				dropout percentile = 0.2
Hidden layer	128	(128,)	(128,)	
Output layer	128	(128,)	(77300,)	activation='softmax'

Table 3.6: Initial LSTM model structure

$$Loss = -\sum_{i=1}^{outputsize} y_i \times \log \hat{y}_i \tag{3.8}$$

Here, $\hat{y}_i$ is the $i-th$ predicted probability value in the model output,y_i is the corresponding target value and the output size is the number of scalar values in the output which is 7730 (number of log types) in our case. With the calculation, the loss value can be a good indicator of how distinguishable these two distributions can be. In our output prediction, $\hat{y}_i$ is the probability that log type i appears and the sum of all $\hat{y}_i$ is 1, which means that exactly one event may occur. In the target vector, there is only one event appears which corresponds to our one-hot-encoding for the log messages.

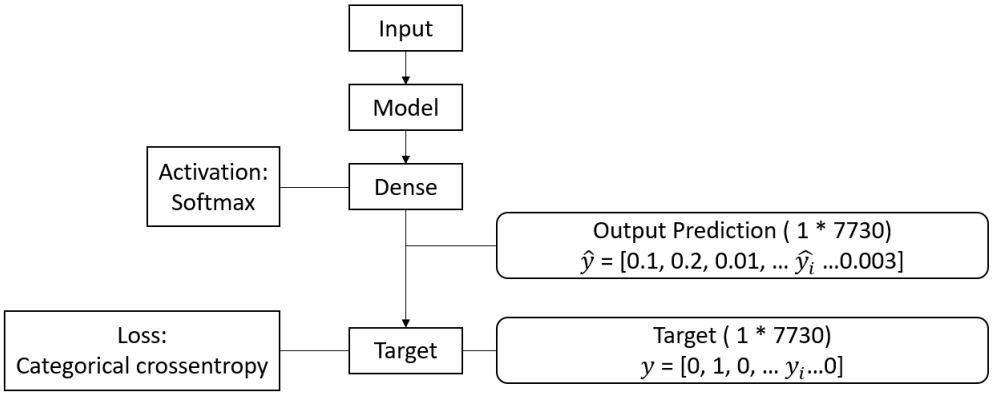

Figure 3.6: Output of LSTM model

Besides, "softmax" is the only recommended activation function to use with categorical crossentropy and thus is chosen. It also ensures that the output of the model will be all positive so that the logarithm of every output value $\hat{y}_i$ exists. The widely used optimizer method – "rmsprop" is chosen in this model. It restricts the oscillations in the vertical direction so that the learning rate can be increased and the algorithm could take larger steps in the horizontal to converge faster. Besides speeding up the mini-batch learning, another advantage of this optimizer is its separate and adaptive learning rate. It adjusts the learning rate by dividing the learning rate for a specific weight by a running average of the magnitudes of recent gradients for that weight[14].

3.3.3 Model tuning and evaluation

The first important thing that can be changed about the LSTM model is structure, which includes the number of layers and the number of neurons in each layer. About the number of layers, Heaton [13] summarized the capabilities of neural network architectures with various hidden layers. According to his theorem, one hidden layer can approximate any function that contains a continuous mapping from one finite space to another and two layers can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy. The more layers added the more complex model will be. In our case, there are only four features of each vector but many instances (240,000 lines per file$\times$ 19 files) to learn. A two-layer architecture is then defined to capture patterns of training data. For the number of neurons, there are many rule-of-thumb methods for determining the correct number of neurons to be used. One of these methods[15] proposes that calculating the number of neurons based on the equation below helps prevent over-fitting. Where N_i stands for the number of input neurons, N_0 stands for the number of output neurons, N_s stands for the number of samples in training data set and α stands for an arbitrary scaling factor which is usually in the range of 2-10.

$$N_h = \frac{N_s}{\alpha \times (N_i + N_0)} \tag{3.9}$$

Based on this equation, with α of 2, we get a result of 144. Empirically, the number of neurons should be the powers of two so the closest number 128 is taken as the number of neurons in our experiment. It can be also tuned to a larger number such as 256 based on performance. Other hyper-parameters like batch size, epoch are tuned according to performance.

To evaluate the validity of the LSTM model, top k categorical accuracy from Keras metrics will be used as one indicator. The top k accuracy metric computes the accuracy rate if the y true value is among the selected top k predictions. Additionally, the same anomaly injection experiment will be run to construct a confusion matrix. Then two confusion matrices will be compared to answer the research question.

3.4 Common practical challenges

3.4.1 Out of memory due to large data set

The transformed (One-hot encoded) matrix for the LSTM model in table 3.5 is very large with 77300 columns and it occupies large disk space and also Random-access memory (RAM) during work, which causes out of memory error. To describe its quantity with real numbers, 19 log files are used as the training set in this study and they request 560 GB disk space to be stored. When it comes to RAM, it is far beyond the machine's capability. Thus, finding solutions to compress the matrix or to process data with many batches.

For compressing the matrix, methods such as feature hashing and sparse matrix are tested in our experiment. Like one-hot encoding, hashing is a way to encode categorical data but fewer dimensions. The intuition behind is to use a hash function and map original data of arbitrary size to fixed-size values, returning hash values and these values are then used as data for training. However, it has collisions when different values (categories) are mapped to the same value. Since it is important in our case to keep each category independent, the hash is tested not to be a good solution. Since the matrix consists of a large number of zeros and only positions with value "1" contain information, the second way to store data in a sparse matrix seems an effective way. The sparse matrix only stores coordinates of non-zero values and it did solve the memory issue. However, the sparse matrices still need to be transferred back to the dense matrix because the Keras model does not accept the sparse matrix as input. This brings us to the second topic, processing data in a streaming way.

Data generator from Keras is such a way that tries to deal with the memory consumption issue. The motivation behind is to break some very large data file into smaller parts and these parts will be subsequently fed to the model for training. The data generator has another advantage that is it allows to generate dataset on multiple cores in real-time and feed it right away to the model, which also helps speed up the process.
Eventually, the solution to solve the memory issue for this study is to combine the sparse matrix with a data generator. First, during data pre-processing stage, the one-hot encoded dataset is stored in the SciPy sparse matrix with which only coordinates of non-zero values are saved. Then when it comes to the training stage,

the data generator helps to transform the sparse matrix back to the dense matrix as the input for the model in streaming. Since it only processes a subset of the whole dataset, it would not fill the whole memory space. Thus, by setting the batch size of data to be processed each time, we are able to control the memory usage based on hardware ability.

3.4.2 Running speed

Running speed is also a challenge when the machine is not powerful enough to deal with a very large data set. In this study, with a 2.8 GHz CPU and no GPU, it takes 18 hours to finish 3 epochs on training a total of 19 log files. When more epochs are added, it takes even much longer. Therefore, how to speed up the process is also an important topic to discuss. There are mainly two ways to improve, first to improve from the programming side and another way is to find more powerful machines to train.

To improve the code efficiency, several rounds of code reviews are done during experiments to better increase efficiency. As the most commonly used python library used for data manipulation and data analysis, library pandas is also used for many places in this study. However, it takes much longer to process data in a data frame format compared to python native formats like array or list. Thus, to improve the performance regarding speed, code refactoring has been done for three rounds during the whole project.

Regarding the hardware, the Graphics Processing Unit (GPU) is one of the most widespread tools to improve computation performance due to its implicit parallel nature[29]. Particularly, GPUs have been applied extensively in data mining algorithms and performing well[4]. Many libraries, such as TensorFlow, can harness the power and speed of computation that GPUs offer through CUDA toolkits(CUDA, a parallel computing platform and application programming interface (API) model created by Nvidia). In this study, the experiment is done with back-end support from TensorFlow and thus can utilize the GPU resource. By shifting to a machine equipped with GPUs, the time to train all file is shortened to 107 mins, which is ten times faster.

3.4.3 Incremental learning

Incremental learning is a method in which the input data is continuously fed to extend the existing model's knowledge. There are two main reasons to apply incremental learning in machine learning. From the computational intelligence perspective, incremental learning is important to help process large-scale dynamic stream data. Second, from the machine intelligence side, intelligent models should be capable to learn information incrementally. This is a wise way for machine intelligence because it can accumulate experience and knowledge without forgetting things learned before[9]. In this paper, due to the updating nature of logs, incremental learning is also applied in the experiment. The final model works in a way that whenever one training session is over, a checkpoint will be saved as the initial status for the next training. Therefore, the model does not train from scratch every time. Instead, it

is trying to update the model based on accumulated knowledge.

Chapter 4

Experiment results

4.1 LSTM

Table 4.5 records the LSTM tuning process with different number of layers, neurons in the hidden layers, and batch size. In this table, the accuracy refers to categorical accuracy which examines if the top one prediction matches the real value. The value in bold refers to the best performance for each column. As can be learned from these values, the last row has the best accuracy value and the training time is relatively short. Thus, the parameter setting of the last row is used as the experiment setting.

Layers	Hidden neurons	Batch size	Accuracy	Time/per file (s)
2	128	512	0.88064	**1794**
3	128	512	0.86798	1878
2	256	512	0.88711	1808
3	256	512	0.88436	2258
2	256	256	0.89052	3492
2	256	1024	**0.89083**	1872

Table 4.1: Experiment setting

LSTM experiment is running with the optimal parameter setting shown in table 4.2. The experiment is done with incremental learning, thus the total 19 training sets

parameter	value
Hidden layer	2
Hidden layer neurons	256
Dropout layer rate	0.2
Epochs	10
Batch size	1024

Table 4.2: Experiment setting

are added one by one to the model. Each time a new log file is added, it will be trained based on weights saved from last round.

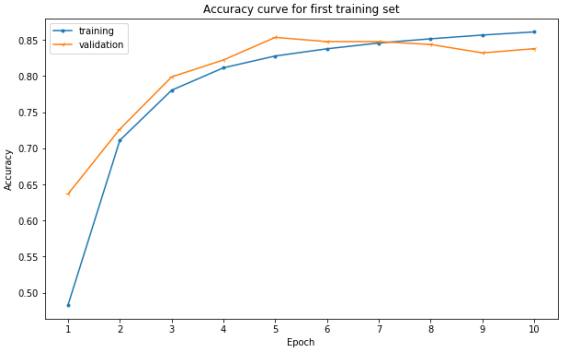

Figure 4.1: Accuracy curve

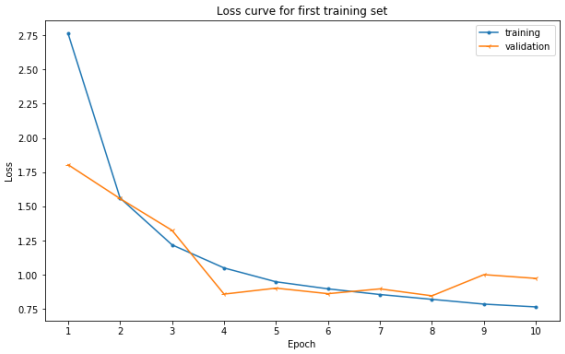

Figure 4.2: Loss curve

Figure 4.3: Accuracy curve

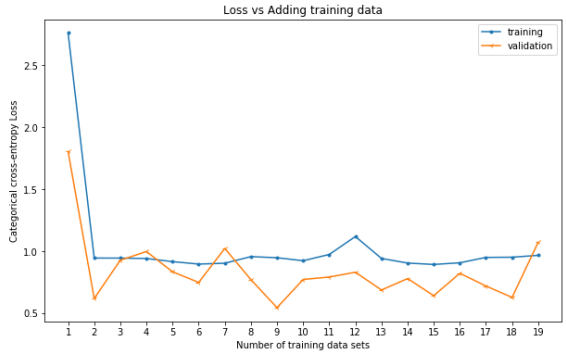

Figure 4.4: Loss curve

Graphs 4.1 and 4.2 below show the training progress for the first single training data set.The blue line represents the training data while orange line stands for validation data.Graph 4.1 shows the accuracy curve of first training data set with 10 epochs. Graph 4.2 shows the loss curve of the first training data set. After 10 epochs, the accuracy for training data set grows from 0.48 to 0.86, and for validation set from 0.63 to 0.84. In these two graphs, the accuracy refers to categorical accuracy, which is the accuracy rate of the top one prediction.

Graphs 4.3 and 4.4 show the progress when adding more data sets to update the model. In these two graphs, X axis represents the number of files that have been added, while Y axis shows the corresponding accuracy and loss value. As can be seen from the graph, accuracy increases rapidly during the first training. However, with adding more files to the model, training accuracy tends to be steady and validation set fluctuates slightly. However, validation set achieves higher accuracy compared with training set surprisingly. The accuracy score for validation set is 0.864 on average, with its best performance reached 0.891 when the 7th file is added.

As the “k” value which determines the accuracy is also an variable, different “k” values is also experimented to compare the performance. Table 4.3 is a comparison of performance with different “k” values. Here, “k” = 5 refers to the case that real log line is compared with top 5 predictions and the accuracy is calculated based on the validation set. As the “k” value grows bigger, the accuracy also increases. And with the top 15 predictions, the accuracy can reach 95.1%.

K value	Accuracy
5	0.93750
10	0.94629
15	0.95117

Table 4.3: Top k prediction accuracy with k = 5,10,15

The rule of detecting anomalies is to categorize those real log lines which are not in the top 10 predictions as abnormal lines. Graph 4.5 shows the anomalies distribution in the test data set, where x axis stands for the line number and y axis for the number of anomalies within the corresponding log range. In total, 12844 lines are detected as anomalies out of 248748 lines, which is 5.16% of the log file. Additionally, as can be seen from the graph most anomalies happen in the beginning or the end of the file, but it appears that not so many anomalies are detected in the middle of the execution.

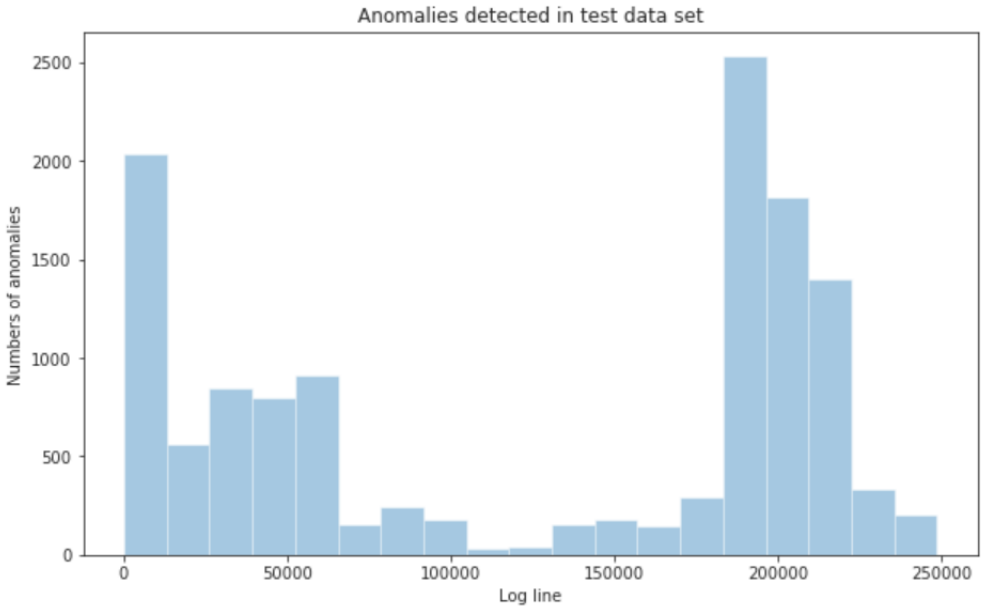

Figure 4.5: Anomalies being detected in test set by LSTM

To also evaluate the LSTM model's performance with real-life cases, anomalies are manually injected into test log files. The operation is done by manually changing the event id randomly in normal log lines. Here, normal lines refers to the log lines that are not detected as anomalies by the LSTM model. The process of injection is: First, the LSTM model is used to predict anomalies in test log file, which detects 12844 lines as anomalies. Then, an assumption is made that other than these abnormal lines, the left of log are normal. Second, 200 lines are chosen as the evaluation set out of all these normal lines. Among these 200 lines, 100 lines are randomly chosen to inject errors by manually changing the event id. Lastly, the same trained LSTM model is used to detect anomalies within these 200 lines. The result is demonstrated in table 4.4.

	Actual positive(Anomalies)	Actual negative
Detected positive	99	18
Detected negative	1	82

Table 4.4: Confusion matrix of LSTM experiment

As can be calculated from table 4.4, the precision of the LSTM model is 84.62% and the recall is 99%. As a harmonic indicator, the F1 score for LSTM reaches 91.24%.

4.2 PCA

To implement PCA model, scikit-learn library is used for this study[27]. Additionally, as the data set of around 5 million data points is huge, incremental PCA is used for its ability to process data in a streaming mode. Four selected variables, namely *Mono Timestamp*, *Transport*, *Priority* and *Log message*, are encoded as explained in section 3.2.1. However, during the first PCA experiment it was shown that the proportion of variance explained by *Timestamp* itself is already more than 99.9%, which causes a huge bias and the model ignores other variables. Therefore, *timestamp* is then excluded and the PCA experiment is done with the left three variables.

During the PCA dimension reduction, 171 components are kept which keep 95.0% of all encoded information. A PCA model is generated based on all training data, graph 4.6 shows the ratio distribution of the top 20 explained variance.

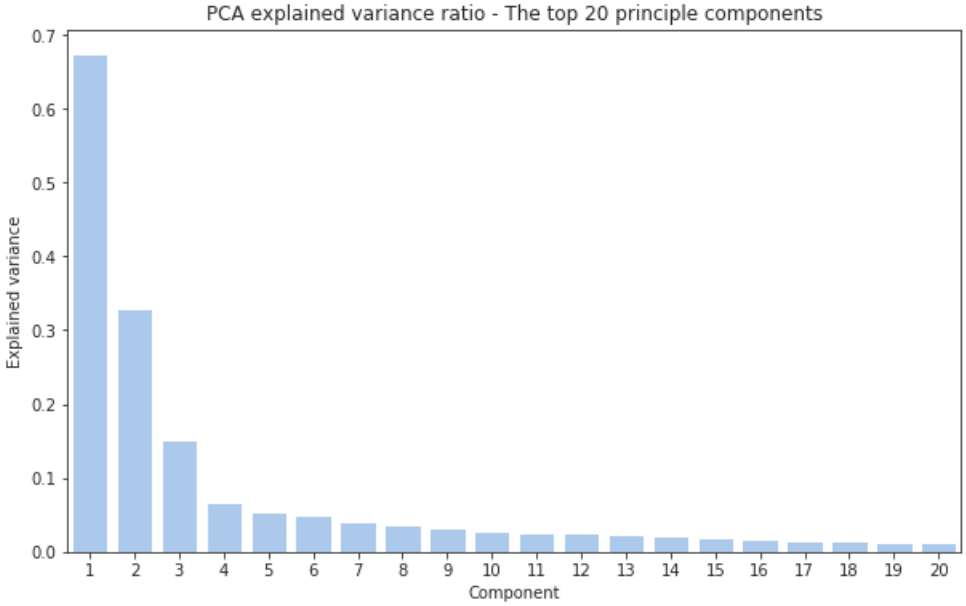

Figure 4.6: Distribution of the top 20 explained variance

Figure 4.7 and 4.8 are two loading plots of PCA components from PC1 to PC4. These plots show how strongly each characteristic influences a principal component. In the loading plots, the project values of each variables on each PC show how much

weight they have on that PC. Besides, the angles between the vectors tell how characteristics correlate with one another.

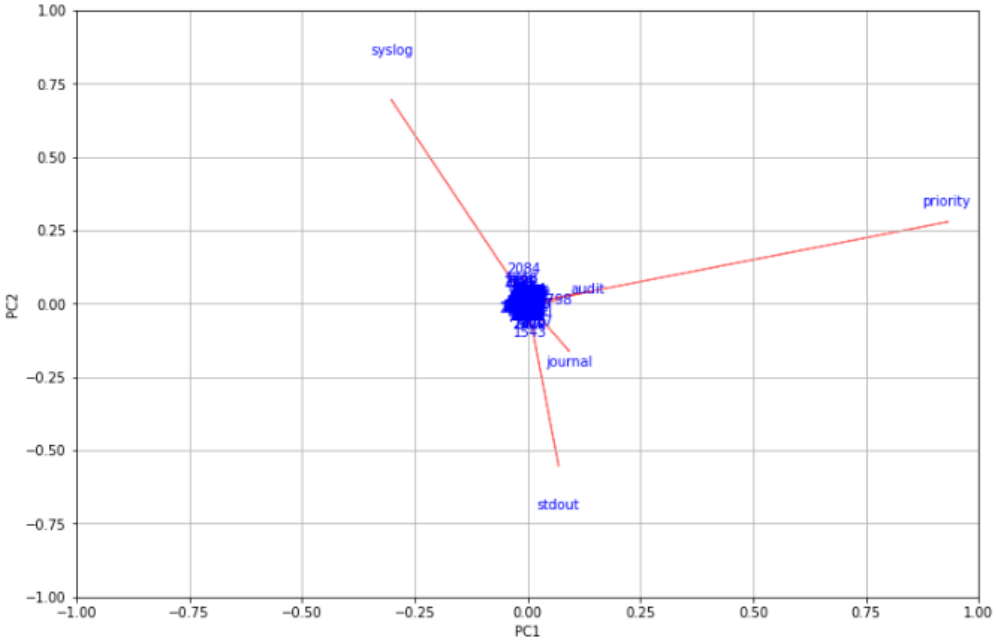

Figure 4.7: PC1 and PC2

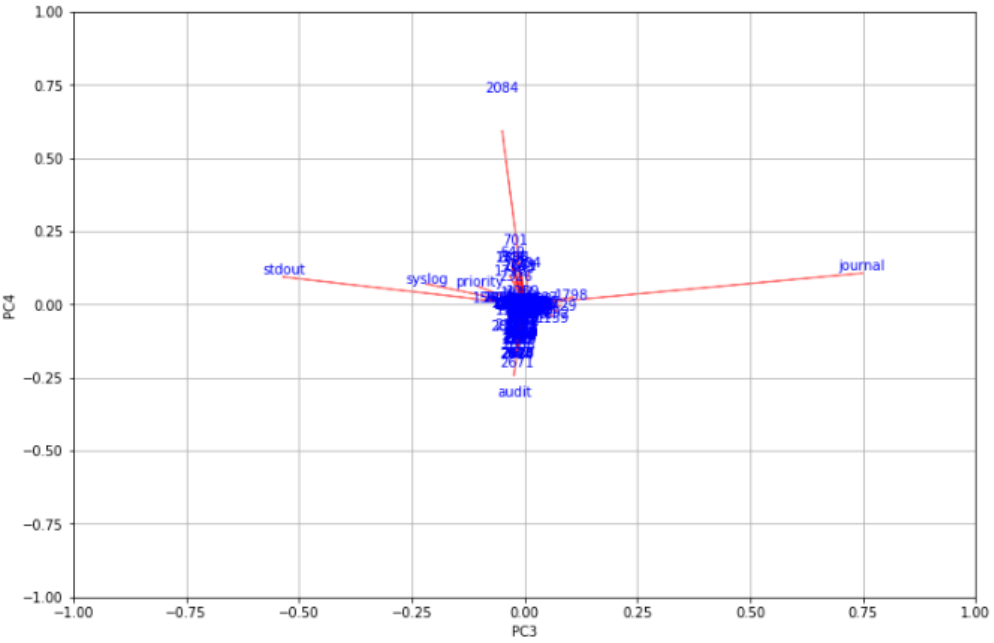

Figure 4.8: PC3 and PC4

As can be seen from the two PCA loading plots, some variables are extremely significant with big projection values. These variables are mainly *priority* and *transport* (e.g. Syslog, stdout, audit) information. About TF-IDF encoded log messages, some important words appear to become significant from PC3 as their projection values become larger. These TF-IDF word variables are numbered from 1-3000, with each number representing one word among the top 3000 important words. For example, "2084", corresponds to a word in TF-IDF vocabulary and shows a signif-

icant contribution to PC4. The result of PCA meets our expectation as it catches the correlation between *transport*, *priority* and *TF-IDF key words* by learning the projection angles and values. This method is interpretable and thus meaningful in real case. As more useful information can be added based on domain knowledge, the PCA model will be sensitive to more comprehensive information.

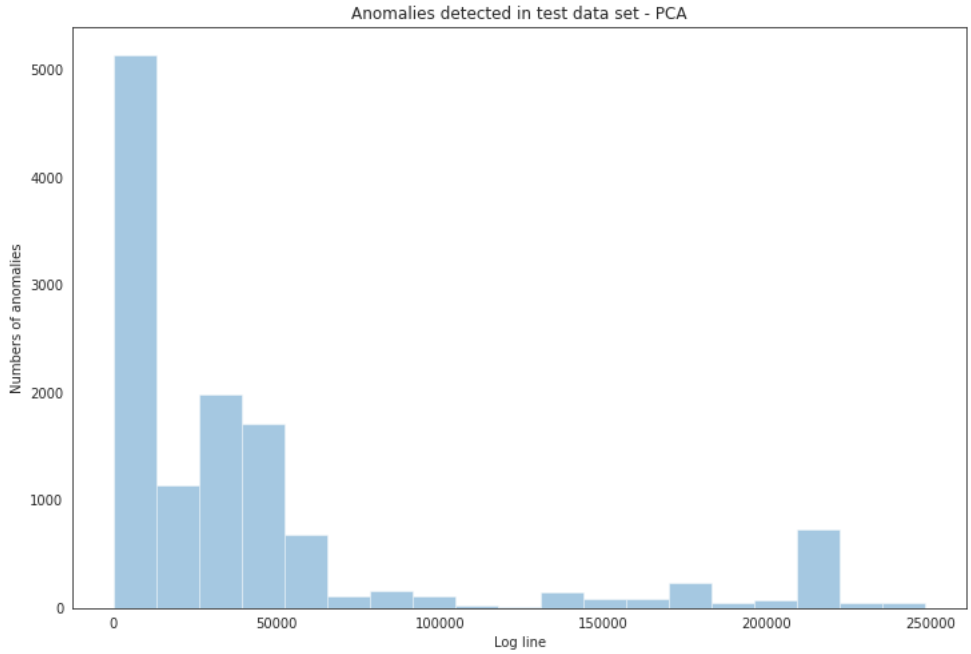

Figure 4.9: Anomalies being detected in test set by PCA

Graph 4.9 illustrates the distribution of anomalies being detected. PCA detects 10768 lines of anomalies out of 248748 lines of log messages, which is 4.33% of the whole log file. Compared with the result of LSTM, these two models both detect many lines as anomalies in the beginning of the log file, not so many in the middle of the log. However, as LSTM detects a significant amount of anomalies in the end of log, PCA does not find many abnormal cases near the tail of log.

To evaluate PCA's performance with real-life cases, two types of anomalies which happen in real life are manually injected into log files. These two types of injection includes wrong *transport* and *priority* information. Consider the situation that we do not have actual labels of each log line, an assumption is made that except for the detected 10768 lines other log messages are labeled as normal in test data set. Then 200 lines are selected randomly from the normal lines in test data set as experiment data set for evaluating. Among these 200 lines, *transport* information is changed manually for 50 lines and *priority* information is changed manually for another 50 lines. The left 100 lines are kept the same, which will be labeled as normal lines in this experiment. These manual injections are all based on empirical research. PCA model trained based on the full 19 training sets is used for detection and its performance is demonstrated in table 4.5 below.

As can be calculated from table 4.5, the precision of PCA model is 96.59% and the recall is 85%. As a harmonic indicator, the F1 score for PCA reaches 90.43%.

	Actual positive(Anomalies)	Actual negative
Detected positive	85	3
Detected negative	15	97

Table 4.5: Confusion matrix of PCA experiment

Notably, during the experiment running it is found that the squared prediction error (SPE) value of all anomalies are significantly higher than SPE value of most normal instances. As can be seen from table 4.6, the calculated threshold is 2.02, mean of all SPE value is 0.83, 75% of SPE value is below 0.79. However, for injected anomaly with *Transport* category, all instances' SPE value is higher than the threshold, which means all *Transport* anomalies are caught by the model. Notably, these anomalies' SPE value is very centralized as they have small standard deviation. About *priority* anomalies, they have a big standard deviation of 6.31689 and the SPE value ranges from 1.61994 to 25.67360. It means all 15 false negatives come from *priority* category and also the PCA model is not very sensitive to value with smaller changes. For example, if the *priority* changes from 7 to 6, it is hard to detect. But if it changes from 7 to 5, then the SPE value will become bigger and thus easier to be recognized. More research should be done regarding the real situation such as how does this value changes usually when an error happens. However, due to resource limitations of this study, this evaluation will be only considered in the future work.

	Full test set	Priority anomalies	Transport anomalies
count	8000	50	50
mean	0.83182	6.97935	2.68798
std	0.63762	6.31689	0.33085
min	0.15766	1.61994	2.28136
25%	0.71005	2.78225	2.56496
50%	0.76222	4.73581	2.67491
75%	0.79152	9.66994	2.73777
max%	25.67360	25.67360	4.75165

Table 4.6: SPE value comparison of PCA experiment

4.3 Baseline and comparison

To compare PCA and LSTM model with current method stupid backoff, these three models are trained with the same 19 data sets and validated by the same log file. The column speed in table 4.7 refers to the training time in hour and the anomaly quantity refers to the number of anomalies being detected in the same log file. As can be seen from table 4.7, Stupid backoff performs the best in the speed category, which finishes training within 0.2 hour. In contrast, LSTM takes 9.5 hour which is significantly longer due to its large amount of computation. These two models

are both working based on the sequence of log messages and LSTM model detects fewer lines as anomalies. In this comparison, it is supposed that the fewer lines are detected as anomalies, the better a model performs since the need from end users is to further narrow down the scope of potential abnormal lines to check. Therefore, it can be concluded that PCA and LSTM both surpass the performance of baseline in anomaly quantity.

Model	Speed	Anomaly quantity
Stupid backoff	0.2	14202
PCA	0.25	10768
LSTM	9.5	12844

Table 4.7: Comparison with baseline model

Besides, the same anomaly injection experiment is conducted with stupid backoff model. The result is shown in table 4.8.

	Actual positive(Anomalies)	Actual negative
Detected positive	92	95
Detected negative	8	5

Table 4.8: Confusion matrix of stupid backoff experiment

The precision and recall of three different models are compared in figure 4.10.

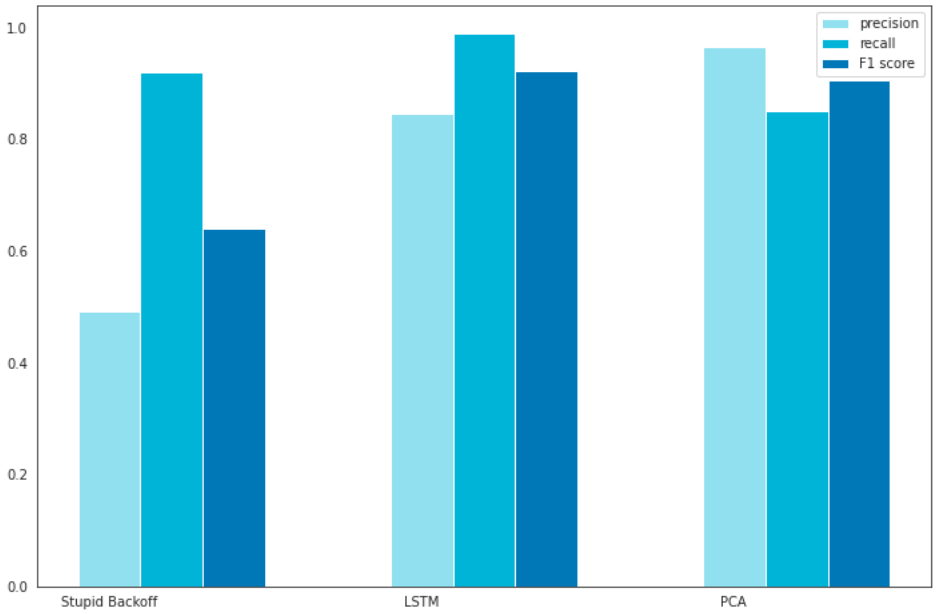

Figure 4.10: Recall, precision and F1 scores comparison

However, the real value of the model is not only about reducing the quantity or catch these “fake” anomalies. It is about truly helping the work of developers, which refers

to the accuracy of shooting real-life trouble. Therefore, a further evaluation should be taken in order to analyse the true value of the model. But due to resource limitations, it will be only discussed in the future work.

Chapter 5

Discussion and Conclusion

This study researches on anomaly detection in two mainstream directions, PCA-based and workflow-based method. Corresponding to these two directions, two models PCA, LSTM are experimented and both performed well.

First, they both successfully narrow down the scope of potential anomalies in the whole log file. LSTM detects 12,844 lines of the failed log as anomalies and PCA detects 10,768 lines, which are by proportion 5.16% and 4.33% of the full log file. By comparing the detected anomaly distributions of these two models, it is found that around 60% of the anomaly lines detected by them are overlapping. Additionally, these two models both show a similar distribution of anomalies that centralize at the head and tail. According to feedback from end-users of this anomaly detection tool, this distribution meets their expectation. To interpret the result explicitly, these lines labeled as anomalies include updating changes and those that violate normal patterns, which are both targets of this study. In this situation, all problems are anomalies, but not all anomalies are problems. Both of the models are trying to detect log lines that are very rare under a specific context, however, it is not promised if these rare cases are problems. But since the need for end-users is to help them narrow down the scope so that they can shoot trouble faster, how to define the real problems is still human's business. The machine's task of finding potential abnormal lines are accomplished and therefore, these two models are regarded as effective in the real scenario. However, to further validate the idea of using these two models. User experience evaluation and iteration are needed until this tool truly benefits end users.

Second, instead of choosing a single model with better performance. The combination of different types of model is recommended for real-life production. As the intuition behind the PCA and LSTM models differ, these two models are suitable for different situations. LSTM model is focusing on the sequence of log messages and therefore it is able to detect the changes of the order of log lines, which can also be understood as workflow pattern. Additionally, as the log parsing phase is done with training sets and thus only log types in training sets will be added to the library of log templates. When a new log type appears in the test data set, it will be categorized as "others" which is also one type of anomaly in this thesis study (updating changes). PCA works for another case with its ability to catch the correlation between different features. It is sensitive to "transport", "priority", and

"keywords in messages", which are three variables encoded during the experiment. Once these three features do not match each other, which will lead to a large SPE value, the instance will be labeled as an anomaly. In practice, PCA will perform well and fit the need for real scenarios with more domain knowledge added. In other words, if typical features that cause anomalies can be summarized based on the experience of engineers, then the PCA model will target these anomalies more accurately. Because the PCA-based method solves the anomaly detection problem by analyzing available features to determine what constitutes a "normal" class [1]. With the assistance of engineers' experience, the available features can be wisely selected because they know what information is important to focus on. Additionally, these features can be encoded accordingly based on their domain knowledge. Take priority level as an example, if one category is significantly more important than others then it could be highlighted with a larger number. In this way, it would enhance the performance of the PCA model.

Overall, it is highly recommended if these two models can be combined and help with different aspects for anomalies detection.

Chapter 6

Future work

6.1 User research and experience evaluation

There is some limitation for this study, especially the way to evaluate model performance. To productize this anomaly detection tool, more end-user feedback should be involved in the iteration. Both implicit methods and explicit methods can be used to evaluate the utility and usability of this application, such as think-aloud protocols, interviews, and system benchmarking. The think-aloud protocol is a method to gather feedback in usability testing in product design and development, it involves participants through thinking aloud as they are asked to perform a set of specific tasks. This method is introduced by Clayton Lewis [20] and it can help dig out the users' real needs. In this anomaly detection tool development, it can also help by observing and listening to how they act in daily debugging, and then this tool will be able to target at their pain point more accurately. Besides, the interview is also a way to collect more in-depth insights from the end-users' daily work. One of the reasons that no standard evaluation is performed in this study is that the real need is not very clear. For example, we are missing the information that what specific types of anomalies happen most frequently, the importance ranking of different types of anomalies, and how these engineers would expect this tool to help. By clarifying these questions, this anomaly detection tool will target more accurately. One last method – system benchmarking, which is already used in this study to compare the results of different models, can be used more widely. During the whole development process, for instance, a profiling tool that benchmarks the computation speed can be added to improve efficiency. Overall, if more user experience methods can be combined in the study of industrial tool development, this tool will benefit users more in real scenarios.

6.2 Adding features

Currently, both models are focusing only on log patterns. However, except for the anomaly identified by abnormal workflow and pattern, another type of anomaly can be ascribed to the abnormal parameters conveyed by the log message. For example, when the IP address should be within the same range of allowable LAN and it appears not to be, it will cause the failure of installation but can not be detected by our model. The reason is that these parameters are filtered out by regular expressions during the log pre-processing phase to extract log type. In order to detect

this type of anomaly, during grouping different types of logs as templates, their parameters should be stored in separate matrices. To deal with these parameters, empirical research is needed to manually differentiate the important parameters and drop the meaningless ones such as generated keys. Then separate models are needed to be built for each specific type of log so that it can not only tell when something goes wrong but also make it explainable (value too big/small, not in a normal range, etc.).

6.3 Algorithm improvement

In this study, two representative algorithms for PCA-based and work-flow based methods are tested, namely PCA and LSTM. They both completed the task to detect anomaly detection with a success rate above 80%. However, these two models, especially the LSTM model, take a long time to finish the task. So there is still a growing space to improve efficiency. Regarding computation speed, the algorithm can be optimized according to the time report produced by profiling tools. Different functions, especially for the data pre-processing step, can be optimized to more efficient and reliable code with the optimal algorithm complexity [26]. Furthermore, more models can be evaluated to be compared with LSTM.

Bibliography

[1] Pca-based anomaly detection - ml studio (classic) - azure — microsoft docs. https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/pca-based-anomaly-detection: :text=The (Accessed on 08/25/2020).

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] BC Brookes. The shannon model of ir systems. *Journal of Documentation*, 1972.

[4] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th international conference on Machine learning*, pages 104–111, 2008.

[5] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media, 2005.

[6] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

[7] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[8] Ricardo Dunia and S Joe Qin. Multi-dimensional fault diagnosis using a subspace approach. In *American Control Conference*, 1997.

[9] Haibo He, Sheng Chen, Kang Li, and Xin Xu. Incremental learning from stream data. *IEEE Transactions on Neural Networks*, 22(12):1901–1914, 2011.

[10] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 654–661. IEEE, 2016.

[11] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[12] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[13] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[14] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8), 2012.

[15] hobs (https://stats.stackexchange.com/users/15974/hobs). How to choose the number of hidden layers and nodes in a feedforward neural network? Cross Validated. URL:https://stats.stackexchange.com/q/136542 (version: 2019-05-27).

[16] J Edward Jackson and Govind S Mudholkar. Control procedures for residuals associated with principal component analysis. *Technometrics*, 21(3):341–349, 1979.

[17] Donald R Jensen and Herbert Solomon. A gaussian approximation to the distribution of a definite quadratic form. *Journal of the American Statistical Association*, 67(340):898–902, 1972.

[18] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[19] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. *ACM SIGCOMM computer communication review*, 34(4):219–230, 2004.

[20] Clayton Lewis. *Using the" thinking-aloud" method in cognitive interface design*. IBM TJ Watson Research Center Yorktown Heights, NY, 1982.

[21] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111. IEEE, 2016.

[22] Chris Lonvick. Rfc3164: The bsd syslog protocol, 2001.

[23] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 613–622, 2010.

[24] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[25] James H Martin and Daniel Jurafsky. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson/Prentice Hall Upper Saddle River, 2009.

[26] Svetlin Nakov and Veselin Kolev. *Fundamentals of Computer Programming with C#: The Bulgarian C# Book*. Faber Publishing, 2013.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.

[29] Raghavendra D Prabhu. Somgpu: an unsupervised pattern classifier on graphical processing unit. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 1011–1018. IEEE, 2008.

[30] D Team. Data block in hdfs - hdfs blocks & data block size - dataflair. https://data-flair.training/blogs/data-block/. (Accessed on 08/25/2020).

[31] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, pages 119–126. IEEE, 2003.

[32] Risto Vaarandi and Mauno Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7. IEEE, 2015.

[33] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[34] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News*, 44(2):489–502, 2016.

[35] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

Appendix

A. Qstatistic: Lookup table for c_alpha

C_alpha	alpha
1.7507	0.08
1.9600	0.05
2.5758	0.01
2.807	0.005
2.9677	0.003
3.2905	0.001
3.4808	0.0005
3.8906	0.0001
4.4172	0.00001